
Manual de Referencia
Cubre Java SE6
Código en linea gratis
mcgraw-hill-educacion.com
JAVA
Séptima edición
• Guía comprensiva para todo el lenguaje Java.
• Cubre applets, servlets, Swing, JavaBeans, AWT y colecciones.
• Cientos de ejemplos y aplicaciones simples.
Mc Graw Hill
Herbert Schildt

Java
Manual de referencia

Acerca del autor

Herbert Schildt Es la máxima autoridad en los lenguajes de programación Java, C, C++ y C#. Sus libros de programación han vendido más de 3.5 millones de copias en todo el mundo y se han traducido a la mayoría de los idiomas. Es autor de *Manual de referencia de C#, Manual de referencia de C, El arte de programar en Java, Fundamentos de Java y Java 2 Manual de referencia* entre otros *best sellers*. Schildt es egresado y posgraduado de la University of Illinois. Se le puede contactar en la oficina de su consultoría, (217) 586-4683. Su sitio Web es: **www.HerbSchildt.com.**

Java
Manual de referencia, Séptima edición

Herbert Schildt

Traducción

Javier González Sánchez
Tecnológico de Monterrey Campus Guadalajara

Rosana Ramos Morales
Universidad de Guadalajara

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

Director Editorial: Fernando Castellanos Rodríguez
Editor de desarrollo: Miguel Ángel Luna Ponce
Supervisora de producción: Jacqueline Brieño Álvarez
Formación: Overprint, S.A. de C.V.

Java Manual de referencia
Séptima edición

Educación

A Subsidiary of The McGraw-Hill Companies, Inc.
Corporativo Punta Santa Fe
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 13: 978-970-10-6288-3
ISBN 10: 970-10-6288-4

Translated from the 7th English edition of
Java: The Complete Reference
By: Herbert Schildt

ISBN-10: 0-07-226385-7
ISBN-13: 978-0-07-226385-5

7890123456 8765432109

Impreso en México Printed in Mexico

Resumen del contenido

Contenido

Prefacio

Mientras escribo esto, Java está justo iniciando su segunda década. A diferencia de muchos otros lenguajes de computadora cuya influencia comienza a disminuir con el paso de los años, la influencia de Java ha crecido fuertemente con el paso del tiempo. Java saltó a la fama como opción para programar aplicaciones en Internet con su primera versión. Cada versión subsiguiente ha solidificado esa posición. Hoy día, Java sigue siendo la primera y mejor opción para desarrollo de aplicaciones Web.

Una de las razones del éxito de Java es su agilidad. Java se ha adaptado rápidamente a los cambios en el ambiente de desarrollo y a los cambios en la forma en que los programadores programan. Y lo más importante, Java no sólo ha seguido las tendencias, ha ayudado a crearlas. A diferencia de muchos lenguajes que tienen un ciclo de revisión de aproximadamente 10 años, en promedio los ciclos de revisión de Java son de alrededor de 1.5 años. La facilidad de Java para adaptarse a los rápidos cambios en el mundo de la computación es una parte crucial del porque ha permanecido a la vanguardia del diseño de lenguajes de programación. Con la versión de Java SE 6, el liderazgo de Java es indiscutible. Si estamos realizando programas para Internet, hemos seleccionado el lenguaje correcto. Java ha sido y continúa siendo el lenguaje más importante para el desarrollo de aplicaciones en Internet

Como muchos lectores sabrán, ésta es la séptima edición del libro, el cual fue publicado por primera vez en 1996. Esta edición ha sido actualizada para Java SE 6. También ha sido extendida en muchas áreas clave, como ejemplo de ello podemos mencionar que ahora se incluye más cobertura de Swing y una discusión más detallada de los paquetes de recursos. De principio a fin hay muchos otros agregados y mejoras. En general, un gran número de páginas con material nuevo han sido incorporadas.

Un libro para todos los programadores

Este libro es para todo tipo de programadores, principiantes y experimentados. Los principiantes encontrarán discusiones cuidadosamente establecidas y ejemplos particularmente útiles. Para el programador experimentado se ha realizado una cobertura profunda de las más avanzadas características de Java y sus bibliotecas. Para ambos, este libro ofrece un recurso duradero y una referencia fácil de utilizar.

Qué contiene

Este libro es una guía completa y detallada del lenguaje de programación Java, describe su sintaxis, palabras clave y principios fundamentales de programación. Además de examinar porciones significativas de las bibliotecas de Java. El libro está divido en cuatro partes, cada una se enfoca en un aspecto diferente del ambiente de programación de Java.

La primera parte presenta un tutorial detallado del lenguaje de programación Java. Comienza con lo básico, incluyendo temas como tipo de datos, sentencias de control y clases. La primera parte también trata el mecanismo de gestión de excepciones de Java, el subsistema de multihilos, los paquetes y las interfaces. Por supuesto las nuevas características de Java, tales como tipos parametrizados, anotaciones, enumeraciones y autoboxing son cubiertas a detalle.

La segunda parte examina aspectos clave de las bibliotecas estándares del API de Java. Los temas que se incluyen son las cadenas de caracteres, la construcción de flujos de E/S, el trabajo en red, las utilerías estándares, la estructura de colecciones, los applets, los controles basados en interfaces gráficas de usuario, las imágenes y la concurrencia.

La tercera parte examina tres importantes tecnologías de Java: Java Beans, Swing y servlets.

La cuarta parte contiene dos capítulos que muestran ejemplos de Java en acción. En el primer capítulo se desarrollan varios applets para realizar cálculos financieros comunes, tales como calcular el pago regular de un préstamo o la inversión mínima necesaria para retirar mensualmente una cantidad determinada. Este capítulo también muestra como convertir esos applets en servlets. El segundo capítulo desarrolla un administrador de descarga de archivos que supervisa dichas descargas. Esta aplicación tiene la habilidad de iniciar, detener, suspender y continuar. Ambos capítulos son adaptaciones de textos tomados de mi libro *The Art of Java*, del cual fui coautor junto con James Holmes.

El código está en la Web

Recuerde que el código fuente, de todos los ejemplos en este libro, está disponible sin costo en la Web en la página **www.mcgraw-hill-educacion.com.**

Agradecimientos

Patrick Naughton merece una mención especial. Patrick fue uno de los creadores del lenguaje Java, y colaboró en la primera edición de este libro. Gran parte del material de los capítulos 19, 20 y 25 fue proporcionado inicialmente por Patrick. Su perspicacia, experiencia y energía contribuyeron en gran medida al gran éxito de este libro.

También agradezco a Joe O'Neil el haberme proporcionado los borradores iniciales de los capítulos 27, 28, 30 y 31. Joe ha colaborado en varios de mis libros y, como siempre, su esfuerzo es apreciado.

Finalmente, muchas gracias a James Holmes por proporcionar el capítulo 32. James es un programador y autor extraordinario. Trabajó conmigo en la escritura de *The Art of Java*, es autor de *Struts The Complete Reference* y uno de los coautores de *JSF: The Complete Reference*.

HERBERT SCHILDT

Referencias adicionales

Este libro es la puerta de acceso a los libros de programación de la serie de Herb Schildt. Algunos otros textos de interés se citan a continuación:

Para aprender más acerca de la programación en Java, recomendamos los siguientes:

Java: A Beginner's Guide

Swing: A Beginner's Guide

The Art of Java

Para aprender acerca de C++, encontrarás especialmente útiles los siguientes libros:

C++: The Complete Reference

C++: A Beginner's Guide

The Art of C++

C++ From the Ground Up

STL Programming From the Ground Up

Para aprender acerca de C#, sugerimos los siguientes libros de Schildt:

C#: The Complete Reference

C#: A Beginner's Guide

Para aprender acerca del lenguaje C, los siguientes títulos serán interesantes:

C: The Complete Reference

Teach Yourself C

Cuando necesite respuestas sólidas y rápidas, diríjase a Herbert Schildt, la autoridad reconocida en el mundo de la programación.

PARTE I

El lenguaje Java

CAPÍTULO 1

Historia y evolución de Java

Para entender completamente Java, se deben entender las razones detrás de su creación, las fuerzas que lo formaron y el legado que hereda. Java es una mezcla de los mejores elementos de los lenguajes de programación exitosos. El resto de los capítulos de este libro describirán los aspectos prácticos de Java incluyendo su sintaxis, bibliotecas principales y aplicaciones. Este capítulo explica cómo y porqué surge Java, qué lo hace tan importante, y cómo se ha desarrollado a través de los años.

Aunque ha sido fuertemente ligado a Internet, es importante recordar que Java es un lenguaje de programación de uso general. Las innovaciones y desarrollo de los lenguajes de programación ocurren por dos razones fundamentales:

- Para adaptarse a los cambios en ambientes y usos
- Para implementar refinamientos y mejoras en el arte de la programación

Como verá, el desarrollo de Java fue dirigido por ambos elementos en similar medida.

Linaje de Java

Java está relacionado con C++, que es un descendiente directo de C. Java hereda la mayor parte de su carácter de estos dos lenguajes. De C, Java deriva su sintaxis y muchas de sus características orientadas a objetos fueron consecuencia de la influencia de C++. Efectivamente, muchas de las características de Java vienen de –o surgen como respuesta a– sus lenguajes predecesores. Más aún, la creación de Java está profundamente arraigada en el proceso de refinamiento y adaptación, que en las pasadas décadas ha ocurrido con los lenguajes de programación. Por estos motivos, en esta sección revisaremos la secuencia de eventos y factores que condujeron a la creación de Java. Como se verá, cada innovación en el diseño de un lenguaje de programación se debe a la necesidad de resolver un problema al que no han podido dar solución los lenguajes precedentes. Java no es una excepción.

El Nacimiento de la programación moderna: C

El lenguaje C sacudió el mundo de la computación. Su repercusión no debería ser subestimada, ya que cambió fundamentalmente la forma en que la programación era enfocada y concebida. La creación de C fue un resultado directo de la necesidad de un lenguaje de alto nivel, estructurado, eficiente y que pudiera reemplazar al código ensamblador en la creación de programas. Como

probablemente sabrá, cuando se diseña un lenguaje de programación se realiza una serie de balances comparativos, tales como:

- Facilidad de uso frente a potencia
- Seguridad frente a eficiencia
- Rigidez frente a extensibilidad

Antes de la aparición de C, los programadores usualmente tenían que elegir entre lenguajes que optimizaran un conjunto de características u otro. Por ejemplo, aunque FORTRAN podía utilizarse para escribir programas muy eficientes en aplicaciones científicas, no resultaba muy bueno para implementar aplicaciones de sistema.Y mientras BASIC era fácil de aprender, no era muy poderoso, y su falta de estructura cuestionaba su utilidad en el desarrollo de programas grandes. El lenguaje ensamblador se puede utilizar para generar programas muy eficientes, pero su aprendizaje y uso no resultan muy sencillos. Además, la depuración del código ensamblador resulta bastante complicada.

Otro problema complejo fue que los primeros lenguajes de computadora como BASIC, COBOL y FORTRAN no fueron diseñados en torno a los principios de la estructuración. En lugar de eso, dependían del GOTO como forma más importante de control de flujo. Como consecuencia, los programas escritos con estos lenguajes tendían a producir "código spaghetti": un código lleno de saltos enredados y ramificaciones condicionales que hacen que la comprensión de un programa resulte virtualmente imposible. Por otro lado, lenguajes estructurados, como Pascal, no fueron diseñados pensando en la eficiencia y fallaron al intentar incluir ciertas características necesarias para hacerlos aplicables en una amplia gama de sistemas; específicamente, dado los dialectos estándares de Pascal disponibles en ese entonces, no era práctico considerar el uso de Pascal para elaborar aplicaciones de sistema.

Así, justo antes de la aparición de C, ningún lenguaje había conseguido reunir los atributos que habían concentrado los primeros esfuerzos. Existía la necesidad de un nuevo lenguaje. A principios de los años setenta tuvo lugar la revolución informática, y la demanda de software superó rápidamente la capacidad de los programadores de producirlo. En los círculos académicos se hizo un gran esfuerzo en un intento de crear un lenguaje de programación mejor que los existentes. Pero y quizás lo más importante, una fuerza secundaria comenzaba a aparecer. El hardware de la computadora se estaba convirtiendo en algo bastante común, de manera que se estaba alcanzando una masa crítica. Por primera vez, los programadores tenían acceso ilimitado a sus máquinas, y esto permitía la libertad de experimentar. Esto también consintió que los programadores comenzaran a crear sus propias herramientas. En la víspera de la creación de C, todo estaba preparado para dar un salto hacia adelante en los lenguajes de programación.

C fue inventado e implementado por primera vez por Dennis Ritchie en una DEC PDP-11 corriendo el sistema operativo UNIX. C fue el resultado del proceso de desarrollo que comenzó con un lenguaje anterior llamado BCPL, desarrollado por Martin Richards. BCPL tenía influencia de un lenguaje llamado B, inventado por Ken Thompson, que condujo al desarrollo de C en la década de los años setenta. Durante muchos años, el estándar para C fue, de hecho, el que era suministrado con el sistema operativo UNIX y descrito en"The C programming Language"por Brian Kernighan y Dennis Ritchie (Prentice-Hall, 1978). C fue formalmente estandarizado en diciembre de 1989, cuando se adoptó el estándar ANSI (American National Standards Institute) de C.

La creación de C es considerada por muchos como el comienzo de la era moderna en los lenguajes de programación. Sintetizaba con éxito los conflictivos atributos que habían causado tantos problemas a los anteriores lenguajes de programación. El resultado fue un lenguaje poderoso, eficiente y estructurado cuyo aprendizaje era relativamente fácil. También tenía

otro aspecto casi intangible: era el lenguaje de los programadores. Antes de la invención de C, los lenguajes de programación eran diseñados generalmente como ejercicios académicos o por comités burocráticos. C es diferente. Fue diseñado, implementado y desarrollado por programadores que reflejaron en él su forma de entender la programación. Sus características fueron concebidas, probadas y perfeccionadas por personas que en realidad usaban el lenguaje. El resultado fue un lenguaje que a los programadores les gustaba utilizar. En efecto, C consiguió rápidamente muchos seguidores quienes tenían en él una fe casi religiosa, y como consecuencia encontró una amplia y rápida aceptación en la comunidad de programadores. En resumen, C es un lenguaje diseñado por y para programadores. Como se verá, Java ha heredado este legado.

El Siguiente Paso: C++

Durante los últimos años de los setenta y principios de los ochenta, C se convirtió en el lenguaje de programación dominante, y aún sigue siendo ampliamente utilizado. Aunque C es un lenguaje exitoso y útil y que sin duda ha triunfado, se necesitaba algo más. A lo largo de la historia de la programación, el aumento en la complejidad de los programas ha conducido a la necesidad de mejorar las formas de manejar esa complejidad. C++ es la respuesta a esa necesidad. Para entender mejor por qué la gestión de la complejidad de los programas ha dado lugar a la creación de C++, consideremos lo siguiente.

La manera de programar ha cambiado dramáticamente desde que se inventó la computadora. Por ejemplo, cuando se inventaron las primeras computadoras, la programación se hacia manualmente conmutando las instrucciones binarias desde el panel frontal. Este enfoque sirvió mientras los programas consistían de unos pocos cientos de instrucciones. Cuando los programas fueron creciendo, surgió el lenguaje ensamblador, con el cual los programadores podían abordar programas más grandes y cada vez más complejos usando representaciones simbólicas de las instrucciones de la máquina. Conforme los programas continuaron creciendo, los lenguajes de alto nivel fueron introducidos para dar al programador más herramientas con las cuales gestionar la complejidad.

El primer lenguaje ampliamente utilizado fue, claro está, FORTRAN. Aunque FORTRAN fue un impresionante primer paso, no es un lenguaje que anime a desarrollar programas claros y fáciles de entender. En los años sesenta nació la programación estructurada. Este es el método de programación que soportan lenguajes como C. El uso de los lenguajes estructurados permite a los programadores escribir, por primera vez, programas de una complejidad moderada con mayor facilidad. De cualquier forma, aún con los métodos de programación estructurada, una vez que un proyecto alcanza cierto tamaño, su complejidad excede la capacidad de manejo del programador. A principios de los ochenta, muchos de los proyectos estaban llevando al enfoque estructurado más allá de sus límites. Para resolver este problema, una nueva forma de programación surgió, llamada programación orientada a objetos (POO). La programación orientada a objetos se discute en detalle después en este libro, pero aquí está una breve definición: POO es una metodología de programación que ayuda a organizar programas complejos mediante el uso de la herencia, encapsulación y polimorfismo.

En resumen, aunque C es uno de los mejores lenguajes de programación en el mundo, su capacidad para gestionar la complejidad tiene un límite. Una vez que el tamaño del programa excede un cierto punto, se vuelve demasiado complejo tanto así que es muy difícil abarcarlo en su totalidad. Aunque el tamaño preciso en el cual ocurre esta diferencia depende de la naturaleza del programa y del programador, siempre hay un límite en el cual un programa se vuelve

imposible de gestionar. C++ agrega características que permiten pasar estos límites, habilita a los programadores a comprender y manejar programas más largos.

C++ fue inventado por Bjarne Stroustrup in 1979, mientras trabajaba en los Laboratorios Bell en Murray Hill, New Jersey. Stroustrup llamó inicialmente al nuevo lenguaje "C con clases". Sin embargo, en 1983 el nombre fue cambiado a C++. C++ es una extensión de C en la que se añaden las características orientadas a objetos. Como C++ se construye sobre la base de C, incluye todas sus características, atributos y ventajas; ésta es la razón de su éxito como lenguaje de programación. La invención de C++ no fue un intento de crear un lenguaje de programación completamente nuevo. En lugar de eso, fue una ampliación de un lenguaje existente y exitoso.

Todo está dispuesto para Java

A finales de los años ochenta y principios de los noventa la programación orientada a objetos usando C++ dominaba. De hecho, por un pequeño instante pareció que los programadores finalmente habían encontrado el lenguaje perfecto. Como C++ había combinado la gran eficiencia y el estilo de C con el paradigma de la programación orientada a objetos, era un lenguaje que podía utilizar para crear una amplia gama de programas. Sin embargo, como en el pasado, surgieron, una vez más, fuerzas que darían lugar a una evolución de los lenguajes de programación. En pocos años, la World Wide Web e Internet alcanzaron una masa crítica. Este evento precipitaría otra revolución en el mundo de la programación.

La creación de Java

Java fue concebido por James Gosling, Patrick Naughton, Chris Warth, Ed Frank, y Mike Sheridan en Sun Microsystems, Inc. en 1991. Tomó 18 meses el desarrollo de la primera versión funcional. Este lenguaje fue llamado inicialmente "Oak", pero fue renombrado como "Java" en 1995. Entre la implementación inicial de Oak en el otoño de 1992 y el anuncio oficial de Java en la primavera de 1995, muchas personas contribuyeron al diseño y evolución del lenguaje. Bill Joy, Artur van Hoff, Jonathan Payne, Frank Yellin, y Tim Lindholm realizaron contribuciones clave para la maduración del prototipo original.

Algo sorprendente es que el impulso inicial para Java no fue Internet, sino la necesidad de un lenguaje de programación que fuera independiente de la plataforma (esto es, arquitectura neutral) un lenguaje que pudiera ser utilizado para crear software que pudiera correr en dispositivos electrodomésticos, como hornos de microondas y controles remoto. Como se puede imaginar, existen muchos tipos diferentes de CPU que se utilizan como controladores. El inconveniente con C y C++ (y la mayoría de los lenguajes) es que están diseñados para ser compilados para un dispositivo específico. Aunque es posible compilar un programa de C++ para casi todo tipo de CPU, hacerlo requiere un compilador de C++ completo para el CPU especificado. El problema es que los compiladores son caros y consumen demasiado tiempo al crearse. Era necesaria una solución fácil y más eficiente. En un intento por encontrar tal solución, Gosling y otros comenzaron a trabajar en el desarrollo de un lenguaje de programación portable, que fuese independiente de la plataforma y que pudiera ser utilizado para producir código capaz de ejecutarse en distintos CPU bajo diferentes entornos. Este esfuerzo condujo en última instancia a la creación de Java.

Mientras se trabajaban distintos aspectos de Java, surgió un segundo, y definitivamente más importante, factor, que jugaría un papel crucial en el futuro de Java. Este factor fue, naturalmente, la World Wide Web. Si el mundo de la Web no se hubiese desarrollado al mismo tiempo que

Java estaba siendo implementado, Java podría haber sido simplemente un lenguaje útil para programación de dispositivos electrónicos. Sin embargo, con la aparición de la World Wide Web, Java fue lanzado a la vanguardia del diseño de lenguajes de programación, porque la Web también demandaba programas que fuesen portables.

Aunque la búsqueda de programas eficientes, portables (independientes de la plataforma), es tan antigua como la propia disciplina de la programación, ha ocupado un lugar secundario en el desarrollo de los lenguajes, debido a problemas cuya solución era más urgente. Por otro parte, la mayoría de las computadoras del mundo se dividen en tres grandes grupos: Intel, Macintosh y UNIX. Por ello, muchos programadores han permanecido dentro de sus fronteras sin la urgente necesidad de un código portable. Sin embargo, con la llegada de Internet y de la Web, el viejo problema de portabilidad resurgió. Después de todo, Internet, consiste en un amplio universo poblado por muchos tipos de computadoras, sistemas operativos y CPU. Incluso aunque muchos tipos diferentes de plataformas se encuentran conectados a Internet, a todos los usuarios les gustaría ejecutar el mismo programa. Lo que fue una vez un problema irritante pero de baja prioridad se ha convertido en una necesidad que requiere máxima atención.

En 1993, para el equipo que estaba diseñando Java resultó obvio que el problema de la portabilidad, que se encontraban con frecuencia cuando creaban código para los controladores, era también el problema que se encontraban al crear código para Internet. Efectivamente, el mismo problema que Java intentaba resolver a pequeña escala estaba también en Internet a gran escala.Y esto hizo que Java cambiara su orientación pasando de ser aplicado a los dispositivos electrónicos de consumo a la programación para Internet. Por esto, aunque la motivación inicial fue la de proporcionar un lenguaje de programación independiente de la arquitectura, ha sido Internet quien finalmente ha conducido al éxito de Java a gran escala.

Como se mencionó anteriormente, Java derivó muchas de sus características de C y C++. Los diseñadores de Java sabían que utilizando la sintaxis de C y repitiendo las características orientadas a objetos de C++ conseguirían que su nuevo lenguaje atrajese a las legiones de programadores experimentados en C/C++. Además de las semejanzas evidentes a primera vista, Java comparte con C y C++ algunos de los atributos que hicieron triunfar a C y C++. En primer lugar, Java fue diseñado, probado y mejorado por programadores de trabajaban en el mundo real. Java es un lenguaje que tiene sus fundamentos en las necesidades y la experiencia de las personas que lo diseñaron. Por este motivo, Java es el lenguaje de los programadores. En segundo lugar, Java es un lenguaje coherente y consistente lógicamente. En tercer lugar, excepto por las restricciones que impone el ambiente de Internet, Java permite al programador un control total. En otras palabras, Java no es un lenguaje de entrenamiento; es un lenguaje para programadores profesionales.

Dadas las semejanzas entre Java y C++, se puede pensar que Java es simplemente "La versión de C++ para Internet"; sin embargo, creer esto sería un gran error. Java tiene diferencias prácticas y filosóficas con C++. Si bien es cierto que Java fue influido por C++, no es una versión mejorada de C++. Por ejemplo, Java no es compatible de ninguna forma con C++. Las semejanzas con C++ son evidentes, y si usted es un programador C++, con Java se sentirá como en casa. Otro punto: Java no fue diseñado para sustituir a C++, sino para resolver un cierto tipo de problemas diferentes a los que resolvía C++, y ambos coexistirán en los años venideros.

Como mencionamos al principio de este capítulo, la evolución de los lenguajes de programación se debe a dos motivos: la adaptación a los cambios del entorno y la introducción de mejoras en el arte de la programación. El cambio de entorno que dio lugar a la aparición de Java fue la necesidad de programas independientes de la plataforma destinados a su distribución

en Internet. Sin embargo, Java también incorpora cambios en la forma en que los programadores plantean el desarrollo de sus programas. Por ejemplo, Java amplió y refinó el paradigma orientado a objetos usado por C++, añadiendo soporte para multihilos, y proporcionando una biblioteca que simplifica el acceso a Internet. En resumen, no fueron las características individuales de Java las que lo hicieron tan notable, sino que fue el lenguaje en su totalidad. Java fue la respuesta perfecta a las demandas del emergente universo de computación distribuida. Java fue a la programación para Internet lo que C fue a la programación de sistemas: una revolucionaria fuerza que cambió el mundo.

La conexión de C#

El alcance y poder de Java continúa presente en el mundo del desarrollo de los lenguajes de programación. Muchas de sus características innovadoras, construcciones y conceptos se han convertido en guía de referencia para cualquier nuevo lenguaje.

Quizás el más importante ejemplo de la influencia de Java es C#. Creado por Microsoft para su plataforma .NET, C# está estrechamente relacionado con Java. Por ejemplo, ambos comparten la misma sintaxis general, soportan programación distribuida y utilizan el mismo modelo de objetos. Existen, claro está, diferencias entre Java y C#, pero en general la apariencia de esos lenguajes es muy similar. Esta influencia de Java en C# es el testimonio más fuerte hasta la fecha de que Java redefinió la forma en que pensamos y utilizamos los lenguajes de programación.

Cómo Java cambió al Internet

Internet ha ayudado a Java a situarse como líder de los lenguajes de programación, y Java recíprocamente ha tenido un profundo efecto sobre Internet. Además de simplificar la programación Web en general, Java innovó con un nuevo tipo de programación para la red llamado applet que cambió la forma en que se concebía el contenido del mundo en línea. Java también solucionó algunos de los problemas más difíciles asociados con el Internet: portabilidad y seguridad. Veamos más de cerca cada uno de éstos.

Java applets

Un *applet* es un tipo especial de programa de Java que es diseñado para ser transmitido por Internet y automáticamente ejecutado por un navegador compatible con Java. Un applet es descargado bajo demanda, sin mayor interacción con el usuario. Si el usuario hace clic a una liga que contiene un applet, el applet será automáticamente descargado y ejecutado en el navegador. Los applets son pequeños programas comúnmente utilizados para desplegar datos proporcionados por el servidor, gestionar entradas del usuario, o proveer funciones simples, tales como una calculadora, que se ejecuta localmente en lugar de en el servidor. En esencia, el applet permite a algunas funcionalidades ser movidas del servidor al cliente.

La creación de los applets cambió la programación para Internet porque expandió el universo de objetos que pueden ser movidos libremente en el ciberespacio. En general, hay dos muy amplias categorías de objetos que son transmitidos entre servidores y clientes: información pasiva y programas activos. Por ejemplo, cuando usted lee su correo electrónico, usted está viendo información pasiva. Incluso cuando usted descarga un programa, el código del programa es sólo información pasiva hasta que usted lo ejecuta. En contraste, los applets son dinámicos, ellos mismos se ejecutan.

Si bien los programas dinámicos en la red son altamente deseados, también es cierto que representan serios problemas en las áreas de seguridad y portabilidad. Un programa que se descarga y ejecuta automáticamente en la computadora del cliente debe ser vigilado para evitar que ocasioné daños. También debe ser capaz de correr sobre ambientes y sistemas operativos diferentes y variados. Veamos un poco más de cerca estos puntos.

Seguridad

Como probablemente ya sabe, cada vez que transfiere un programa a su computadora corre un riesgo, porque el código que usted está descargado podría contener virus, caballos de Troya o algún otro código malicioso. El núcleo del problema es que ese código malicioso puede causar daños porque está obteniendo acceso no autorizado a los recursos del sistema. Por ejemplo, un virus podría obtener información privada, como los números de las tarjetas de crédito, estados de cuenta bancarios y claves de acceso realizando una búsqueda en el sistema de archivos de su computadora. Para garantizar que un applet de Java pueda ser descargado y ejecutado en la computadora del cliente con seguridad, fue necesario evitar que un applet pudiera realizar ese tipo de acciones. Para ello Java confina a los applets a ser ejecutados en un ambiente controlado sin permitirle el acceso a los recursos completos de la computadora (pronto veremos cómo se logra esto). La posibilidad de descargar applets con la certeza de que no harán ningún daño y que no producirán violaciones en la seguridad del sistema es considerada por muchos la característica más innovadora de Java.

Portabilidad

La portabilidad es uno de los aspectos más importantes en Internet debido a la existencia de muchos y diferentes tipos de computadoras y sistemas operativos conectados a ésta. Si un programa de Java va a ser ejecutado sobre cualquier computadora conectada a la red, es necesario que exista alguna forma de habilitar al programa para que se ejecute en diferentes sistemas. Por ejemplo, en el caso de un applet, el mismo applet debe ser capaz de ser descargado y ejecutado por una amplia variedad de CPU, sistemas operativos y navegadores conectados a Internet. No es práctico tener diferentes versiones del applet para diferentes computadoras. El *mismo* código debe funcionar en *todas* las computadoras. Es necesario generar código ejecutable portable. Como veremos pronto, el mismo mecanismo que ayuda a garantizar la ejecución segura del applet también ayuda a hacer del applet un código portable.

La magia de Java: el bytecode

La clave que permite a Java resolver ambos problemas, el de la seguridad y el de la portabilidad, es que la salida del compilador de Java no es un código ejecutable, sino un bytecode. El *bytecode* es un conjunto de instrucciones altamente optimizado diseñado para ser ejecutado por una máquina virtual la cual es llamada *Java Virtual Machine* (*JVM,* por sus siglas en inglés). En esencia, la máquina virtual original fue diseñada como un *intérprete de bytecode*. Esto puede resultar un poco sorprendente dado que muchos lenguajes de programación modernos están diseñados para ser compilados en código ejecutable pensando en lograr el mejor rendimiento. No obstante, el hecho de que un programa en Java es ejecutado por la JVM ayuda a resolver los problemas asociados con los programas basados en Web. Veamos por qué.

Traducir un programa Java en bytecode hace que su ejecución en una gran variedad de entornos resulte mucho más sencilla, y la razón es que para cada plataforma, sólo es necesario implementar el intérprete de Java. Una vez que el sistema de ejecución existe para un ambiente

determinado, cualquier programa de Java puede ejecutarse en esa plataforma. Recuerde que, aunque los detalles de la JVM difieran de plataforma a plataforma, todas entienden el mismo Java bytecode. Si Java fuera un lenguaje compilado a un código nativo, entonces versiones diferentes del mismo programa deberían compilarse para cada tipo de CPU conectado al Internet. Obviamente esa solución no es factible. Además, la ejecución del bytecode a través de la JVM es la manera más fácil de crear código auténticamente portable.

El hecho de que Java sea interpretado también ayuda a hacerlo seguro. Como la ejecución de cada programa de Java está bajo el control de la JVM, ésta puede contener al programa e impedir que se generen efectos no deseados en el resto del sistema. Como se verá más adelante, ciertas restricciones que existen en Java, sirven para mejorar la seguridad.

En general, cuando un programa es compilado a una forma intermedia y luego interpretado por una máquina virtual, el programa se ejecuta más lento que si fuese compilado a código nativo; sin embargo, en Java esta diferencia no es tan grande. El bytecode ha sido altamente optimizado para habilitar a la JVM a ejecutar los programas más rápido de lo que se podría esperar.

Aunque Java fue diseñado como un lenguaje interpretado, no hay nada que impida la compilación del bytecode en código nativo para incrementar el rendimiento. Por esta razón, Sun comenzó a distribuir su tecnología HotSpot no mucho tiempo después del lanzamiento inicial de Java. HotSpot proporciona un compilador de bytecode a código nativo denominado Just-in-Time o simplemente JIT por sus siglas en inglés. Cuando un compilador JIT es parte de la JVM, porciones de bytecode son compiladas en código ejecutable en tiempo real sobre un esquema de pieza por pieza. Es importante entender que no es práctico compilar un programa de Java completo en código ejecutable, todo de una sola vez, porque Java realiza varias revisiones en tiempo de ejecución que no podrían ser realizados. Un compilador JIT compila código conforme va siendo necesario, durante la ejecución. Incluso aplicando compilación dinámica al bytecode, la portabilidad y las características de seguridad permanecen debido a que la JVM permanece a cargo del ambiente de ejecución.

Servlets: Java en el lado del servidor

Los applets sin duda son de gran utilidad, sin embargo representan apenas la mitad de la ecuación de los sistemas cliente/servidor. Poco tiempo después del lanzamiento inicial de Java resultó obvio que Java también sería útil en el lado del servidor, para ello se crearon los *servlets*. Un *servlet* es un pequeño programa que se ejecuta en el servidor. De la misma forma que los applets extienden dinámicamente la funcionalidad del navegador Web, los servlets extienden la del servidor Web. Con la aparición de los servlets, Java se posicionó como un lenguaje de programación útil en ambos lados de los sistemas cliente/servidor.

Los servlets son utilizados para enviar al cliente contenido que es creado y generado dinámicamente. Por ejemplo, una tienda en línea podría usar un servlet para buscar el precio de un artículo en una base de datos. La información obtenida de la base de datos puede ser utilizada para construir dinámicamente una página Web que es enviada al navegador del cliente que solicitó la información. Si bien existen diversos mecanismos para generar contenido de manera dinámica en el Web, tales como CGI (Common Gateway Interface), los servlets ofrecen diversas ventajas, entre ellas un mejor rendimiento.

Los servlets son altamente portables debido a que como todos los programas de Java son compilados a bytecode y ejecutados por una máquina virtual, esto garantiza que el mismo servlet pueda ser utilizado en diferentes servidores. Los únicos requerimientos son que el servidor cuente con una JVM y un contenedor de servlets.

Las cualidades de Java

Ninguna discusión sobre la historia de Java está completa sin tener en cuenta las cualidades que describen a Java. Aunque las razones fundamentales de la invención de Java fueron la portabilidad y la seguridad, existen otros factores que también desempeñaron un papel importante en el modelado de la forma final del lenguaje. Las consideraciones clave fueron resumidas por el equipo de Java en la siguiente lista de términos:

- Simple
- Seguro
- Portable
- Orientado a objetos
- Robusto
- Multihilos
- Arquitectura neutral
- Interpretado
- Alto rendimiento
- Distribuido
- Dinámico

Simple

Java fue diseñado con la finalidad de que su aprendizaje y utilización resultaran sencillos para el programador profesional. Contando con alguna experiencia en programación es fácil dominar Java. Si ya se comprenden los conceptos básicos de programación orientada a objetos, aprender Java será aún más sencillo. Lo mejor de todo, si se tiene experiencia programando con C++, cambiar a Java requiere sólo un poco de esfuerzo. La mayoría de los programadores de C/C++ no tienen prácticamente ningún problema al aprender Java porque Java hereda la sintaxis y muchas de las características orientadas a objetos de C++.

Orientado a objetos

Aunque influido por sus predecesores, Java no fue diseñado para tener un código compatible con cualquier otro lenguaje. Esto dio la libertad al equipo de Java de partir de cero. Una consecuencia de esto fue una aproximación clara, pragmática y aprovechable de los objetos. Java ha tomado prestadas muchas ideas de entornos de orientación a objetos de las últimas décadas, logrando un equilibrio razonable entre el modelo purista "todo es un objeto" y el modelo pragmático "mantente fuera de mi camino". El modelo de objetos en Java es sencillo y de fácil ampliación, mientras que los tipos primitivos como los enteros, se mantienen como "no objetos"de alto rendimiento.

Robusto

El ambiente multiplataforma de la Web es muy exigente con un programa, ya que éste debe ejecutarse de forma fiable en una gran variedad de sistemas. Por este motivo, la capacidad para crear programas robustos tuvo una alta prioridad en el diseño de Java. Para ganar fiabilidad, Java restringe al programador en algunas áreas clave, con ello se consigue encontrar rápidamente los errores en el desarrollo del programa. Al mismo tiempo, Java lo libera de tener que preocuparse por las causas más comunes de errores de programación. Como Java es un lenguaje estrictamente tipificado, comprueba el código durante la compilación. Sin embargo, también comprueba el código durante la ejecución. De hecho en Java es imposible que se produzcan situaciones en las que aparecen a menudo errores difíciles de localizar. Una característica clave de Java es que se conoce que el programa se comportará de una manera predecible en diversas condiciones.

Para comprender la robustez de Java, consideremos dos de las causas de fallo de programa más importantes: la gestión de memoria y las condiciones de excepción no controladas (errores en tiempo de ejecución). La gestión de la memoria puede convertirse en una tarea difícil y tediosa en los entornos de programación tradicionales. Por ejemplo en C/C++ el programador

debe reservar y liberar la memoria dinámica en forma manual. Esto puede ocasionar problemas, ya que en ocasiones los programadores olvidan liberar memoria que ha sido reservada previamente o, peor aún, intentan liberar memoria que otra parte de su código todavía está utilizando. Java elimina virtualmente este problema, ya que se encarga en lo interno tanto de reservar la memoria como de liberarla. De hecho, la liberación es completamente automática, ya que Java dispone del sistema de recolección de basura que se encarga de los objetos que ya no se utilizan. En los entornos tradicionales, las excepciones surgen, a menudo, en situaciones tales como la división entre cero, o "archivo no encontrado", y se deben gestionar mediante construcciones torpes y difíciles de leer. En esta área, Java proporciona la gestión de excepciones orientada a objetos. En un programa de Java correctamente escrito, todos los errores de ejecución pueden y deben ser gestionados por el programa.

Multihilo

Java fue diseñado para satisfacer los requisitos del mundo real, de crear programas en red interactivos. Para ello, Java proporciona la programación multihilo que permite la escritura de programas que hagan varias cosas simultáneamente. El intérprete de Java dispone de una solución elegante y sofisticada para la sincronización de múltiples procesos que permiten construir fácilmente sistemas interactivos. El método multihilo de Java, de utilización sencilla, permite ocuparse sólo del comportamiento específico del programa, en lugar de pensar en el sistema multitarea.

Arquitectura neutral

Una cuestión importante para los diseñadores de Java era la relativa a la longevidad y portabilidad del código. Uno de los principales problemas a los que se enfrentan los programadores es que no tienen garantía de que el programa que escriben hoy podrá ejecutarse mañana, incluso en la misma máquina. Las actualizaciones de los sistemas operativos y los procesadores, y los cambios en los recursos básicos del sistema, conjuntamente, pueden hacer que un programa funcione mal. Los diseñadores de Java tomaron decisiones difíciles en el lenguaje y en el intérprete Java en un intento de cambiar esta situación. Su meta fue "escribir una vez; ejecutar en cualquier sitio, en cualquier momento y para siempre". Ese objetivo se consiguió en gran parte.

Interpretado y de alto rendimiento

Como antes se ha descrito, Java permite la creación de programas que pueden ejecutarse en diferentes plataformas por medio de la compilación en una representación intermedia llamada código bytecode. Este código puede ser interpretado en cualquier sistema que tenga un intérprete Java. Como ya se explicó el bytecode fue cuidadosamente diseñado para que fuera fácil de traducir al código nativo y poder conseguir así un rendimiento alto utilizando la característica de JIT. Los intérpretes de Java que proporcionan esta característica no pierden ninguna de las ventajas de un código independiente de la plataforma.

Distribuido

Java fue ideado para el entorno distribuido de Internet, ya que gestiona los protocolos TCP/IP. De hecho, acceder a un recurso utilizando un URL no es muy distinto a acceder a un archivo. Java soporta *invocación remota de métodos* (*RMI,* por sus siglas en inglés). Esta característica permite a un programa invocar métodos de objetos situados en computadoras diferentes a través de la red.

Dinámico

Los programas de Java se transportan con cierta cantidad de información que se utiliza para verificar y resolver el acceso a objetos en el tiempo de ejecución. Esto permite enlazar el código dinámicamente de una forma segura y viable. Esto es crucial para la robustez del entorno de Java, en el que pequeños fragmentos de bytecode pueden ser actualizados dinámicamente en un sistema que está ejecutándose.

La evolución de Java

La versión inicial de Java aún y cuando fue revolucionaria no marcó el fin de la era innovadora de Java.

A diferencia de otros lenguajes de programación que normalmente se van estableciendo a base de pequeñas mejoras incrementales, Java ha continuado evolucionando a un ritmo explosivo. Poco después de la versión 1.0, los diseñadores ya habían creado la versión 1.1. Java 1.1 incorporaba muchos elementos nuevos en sus bibliotecas, redefinía la forma en que los eventos eran gestionados y reconfiguraba muchas características de la biblioteca 1.0. También declaraba obsoletas algunas de las características definidas por Java 1.0. Por lo tanto, Java 1.1 añadía y eliminaba atributos de su versión original.

La siguiente versión fue Java 2, donde el "2" indicaba "segunda generación". La creación de Java 2 fue un parte aguas que marcaba el comienzo de la "era moderna" de este lenguaje de programación que evolucionaba rápidamente. La primera versión de Java 2 tenía asignado el número de versión 1.2, cosa que puede resultar extraña. La razón es que inicialmente se refería a las bibliotecas de Java, pero se generalizó como referencia al bloque completo. Con Java 2 la empresa Sun re-etiquetó a Java como J2SE (Java 2 Plataform Standard Edition) y la numeración de versiones continuó aplicándose ahora con este nombre de producto.

Java 2 añadía nuevas facilidades, tales como los componentes Swing y la estructura de colecciones, además mejoraba la máquina virtual y varias herramientas de programación. También declaraba obsoletos algunos elementos. Los más importantes afectaban a la clase **Thread**, en la que se declaraban como obsoletos los métodos **suspend( ), resume( ),** y **stop( ).**

J2SE 1.3 fue la primera gran actualización de Java 2. En su mayor parte añade funcionalidad y "estrecha" el entorno de desarrollo. En general, los programas escritos para la versión 1.2 y los escritos para la versión 1.3 son compatibles. Aunque la versión 1.3 contiene un conjunto de cambios más pequeño que las versiones anteriores, estos cambios son, no obstante, importantes.

La versión J2SE 1.4 trae consigo nuevas y modernas características. Esta versión contenía varias actualizaciones, mejoras y adiciones importantes. Por ejemplo, agregó la nueva palabra clave **assert**, excepciones encadenadas, y un subsistema basado en canales para E/S. También realizó cambios a la estructura de colecciones y a las clases para trabajo en red. Así como numerosos cambios pequeños realizados en todas partes. Aún con la significativa cantidad de nuevas características, la versión 1.4 mantuvo casi 100 por ciento de compatibilidad con versiones anteriores.

La siguiente versión de Java fue J2SE 5, y fue revolucionaria. De manera diferente a la mayoría de las mejoras anteriores, que ofrecieron mejoras importantes, pero controladas, J2SE 5 fundamentalmente expandió el alcance, poder y rango de acción del lenguaje. Para apreciar la magnitud de los cambios que J2SE 5 realizó a Java, veamos la siguiente lista de nuevas características:

- Tipos parametrizados
- Anotaciones
- Autoboxing y auto-unboxing
- Enumeraciones
- Nueva estructura de control iterativa
- Argumentos variables
- Importación estática
- E/S con formato
- Utilerías para trabajo concurrente

Éstos no son anexos menores o actualizaciones. Cada una de estas características representa una adición significativa al lenguaje. Los tipos parametrizados, la nueva estructura de control iterativa y los argumentos variables introducen nuevos elementos en la sintaxis del lenguaje. Autoboxing y auto-unboxing alteran la semántica del lenguaje. Mientras que las anotaciones añaden una nueva dimensión a la programación. La repercución de estas nuevas características va más allá de sus efectos directos. Estos elementos cambiaron la estructura (cualidades y características) distintivas de Java.

El número de versión siguiente para Java habría sido normalmente 1.5. Sin embargo, las nuevas características eran tan significativas que un cambio de 1.4 a 1.5 no habría expresado la magnitud del cambio. Sun decidió aumentar el número de versión a 5 como una forma de enfatizar que ocurría un acontecimiento importante. Así, la nueva versión de Java fue nombrada J2SE 5, y las herramientas de desarrollo fueron nombradas JDK 5 (por las siglas en inglés de Java Development Kit). Sin embargo, a fin de mantener la consistencia, Sun decidió utilizar 1.5 como el número de versión interno, que también es conocido como el número de versión del desarrollador en contraparte con el "5" en J2SE 5 que es conocido como el número de versión del producto.

Java SE 6

El más reciente lanzamiento de Java se llama Java SE 6, el material en este libro ha sido actualizado para cubrir esta versión. Con el lanzamiento de Java SE 6, Sun una vez más decidió cambiar el nombre de Java. Primero nótese que el "2" ha sido eliminado, así que ahora el nombre es Java SE y el nombre oficial del producto es Java Plataform, Standard Edition 6. Al igual que con J2SE 5, el 6 en Java SE 6 es el número de versión del producto. El número de versión interno o número de versión del desarrollador es 1.6.

Java SE 6 está construido sobre la base de J2SE 5 y añade algunas mejoras. Java SE 6 no agrega ninguna característica impactante al lenguaje Java propiamente, sin embargo incrementa la cantidad de bibliotecas en el API del lenguaje y realiza mejoras en el tiempo de ejecución. En lo que respecta a este libro, los cambios en el núcleo de bibliotecas del lenguaje son los más notables en Java SE 6. Muchos paquetes tienen nuevas clases y muchas de las clases tienen nuevos métodos. Estos cambios se muestran a lo largo del libro. El lanzamiento de Java SE 6 contribuye a solidificar aún más los avances hechos por J2SE 5.

Una cultura de innovación

Desde sus inicios, Java ha estado en el centro de la innovación. Su versión original redefinió la programación para Internet. La máquina virtual de Java (JVM) y el bytecode cambiaron la forma en que concebimos la seguridad y la portabilidad. El applet (y después el servlet) le dieron vida al Web. Los procesos de la comunidad Java (JCP por sus siglas en inglés) redefinió la forma en que las nuevas ideas se asimilan e integran a un lenguaje. El mundo de Java siempre está en constante movimiento y Java SE 6 es la versión más reciente producida en la dinámica historia de Java.

CAPÍTULO 2

Introducción a Java

Como ocurre en otros lenguajes de programación, los elementos de Java no existen de forma aislada, sino que trabajan conjuntamente para conformar el lenguaje como un todo. Sin embargo, esta interrelación puede hacer difícil describir un aspecto de Java sin involucrar a otros. A menudo, una discusión sobre una determinada característica implica un conocimiento anterior de otra. Por esta razón, este capítulo presenta una descripción rápida de varias características claves de Java. El material aquí descrito le proporcionará una base que le permitirá escribir y comprender programas sencillos. La mayoría de los temas que se discuten se examinarán con más detalle en el resto de los capítulos de la primera parte.

Programación orientada a objetos

La programación orientada a objetos (POO) es la base de Java. De hecho, todos los programas de Java están por lo menos a un cierto grado orientados a objetos. POO es tan importante en Java que es mejor entender sus principios básicos antes de empezar a escribir, incluso, programas sencillos en Java. Por este motivo, este capítulo comienza con una discusión sobre aspectos teóricos de POO.

Dos paradigmas

Todos los programas consisten en dos elementos: código y datos. Además, un programa puede estar conceptualmente organizado en torno a su código o en torno a sus datos, es decir, algunos programas están escritos en función de "lo que está ocurriendo" y otros en función de "quién está siendo afectado". Éstos son los dos paradigmas que gobiernan la forma en que se construye un programa. La primera de estas dos formas se denomina *modelo orientado al proceso*. Este enfoque describe un programa como una serie de pasos lineales (es decir, un código). Se puede considerar al modelo orientado al proceso como un *código que actúa sobre los datos*. Los lenguajes basados en procesos, como C, emplean este modelo con un éxito considerable. Sin embargo, como se menciona en el Capítulo 1, bajo este enfoque surgen problemas a medida que se escriben programas más largos y más complejos.

El segundo enfoque, denominado *programación orientada a objetos,* fue concebido para abordar esta creciente complejidad. La programación orientada a objetos organiza un programa alrededor de sus datos (es decir, objetos), y de un conjunto de interfaces bien definidas para esos datos. Un programa orientado a objetos se puede definir como un *conjunto de datos que controlan el acceso al código*. Como se verá, con este enfoque se pueden conseguir varias ventajas desde el punto de vista de la organización.

Abstracción

Un elemento esencial de la programación orientada a objetos es la *abstracción*. Los seres humanos abordan la complejidad mediante la abstracción. Por ejemplo, no consideramos a un coche como un conjunto de diez mil partes individuales, sino que pensamos en él como un objeto correctamente definido y con un comportamiento determinado. Esta abstracción nos permite utilizar el coche para ir al mercado sin estar agobiados por la complejidad de las partes que lo forman. Podemos ignorar los detalles de cómo funcionan el motor, la transmisión o los frenos, y, en su lugar, utilizar libremente el objeto como un todo.

Una forma adecuada de utilizar la abstracción es mediante el uso de clasificaciones jerárquicas. Esto permitirá dividir en niveles la semántica de sistemas complejos, descomponiéndolos en partes más manejables. Desde fuera, el coche es un objeto simple. Una vez en su interior, se puede comprobar que está formado por varios subsistemas: la dirección, los frenos, el equipo de sonido, los cinturones, la calefacción, el teléfono móvil, etc. A su vez, cada uno de estos subsistemas está compuesto por unidades más especializadas. Por ejemplo, el equipo de sonido está formado por un radio, un reproductor de CD y/o un reproductor de cinta. La cuestión es controlar la complejidad del coche (o de cualquier otro sistema complejo) mediante la utilización de abstracciones jerárquicas.

Las abstracciones jerárquicas de sistemas complejos se pueden aplicar también a los programas de computadora. Los datos de los programas tradicionales orientados a proceso se pueden transformar mediante la abstracción en objetos. La secuencia de pasos de un proceso se puede convertir en una colección de mensajes entre estos objetos. Así, cada uno de esos objetos describe su comportamiento propio y único. Se puede tratar estos objetos como entidades que responden a los mensajes que les ordenan *hacer algo*. Ésta es la esencia de la programación orientada a objetos.

Los conceptos orientados a objetos forman el corazón de Java y la base de la comprensión humana. Es importante comprender bien cómo se trasladan estos conceptos a los programas. Como se verá, la programación orientada a objetos es un paradigma potente y natural para crear programas que sobrevivan a los inevitables cambios que acompañan al ciclo de vida de cualquier proyecto importante de software, incluida su concepción, crecimiento y envejecimiento. Por ejemplo, una vez que se tienen objetos bien definidos e interfaces, para esos objetos, limpias y fiables, se pueden extraer o reemplazar partes de un sistema antiguo sin ningún temor.

Los tres principios de la programación orientada a objetos

Todos los lenguajes orientados a objetos proporcionan los mecanismos que ayudan a implementar el modelo orientado a objetos. Estos mecanismos son encapsulación, herencia y polimorfismo. Veamos a continuación cada uno de estos conceptos.

Encapsulación

La encapsulación es el mecanismo que permite unir el código junto con los datos que manipula, y mantiene a ambos a salvo de las interferencias exteriores y de un uso indebido. Una forma de ver el encapsulado es como una envoltura protectora que impide un acceso arbitrario al código y los datos desde un código exterior a la envoltura. El acceso al código y los datos en el interior de la envoltura es estrictamente controlado a través de una interfaz correctamente definida. Para establecer una semejanza con el mundo real, consideremos la transmisión automática de un automóvil. Ésta encapsula cientos de bits de información sobre el motor, como por ejemplo la aceleración, la superficie sobre la que se encuentra el coche y la posición de la palanca de cambios. El usuario tiene una única forma de actuar sobre este complejo encapsulado: moviendo

la palanca de cambios. No se puede actuar sobre la transmisión utilizando las intermitentes o el limpiaparabrisas. Por lo tanto, la palanca de cambios es una interfaz bien definida (de hecho la única) para interactuar con la transmisión. Además, lo que ocurra dentro de la transmisión no afecta a objetos exteriores a la misma. Por ejemplo, al cambiar de marcha no se encienden las luces. Como la transmisión está encapsulada, docenas de fabricantes de coches pueden implementarla de la forma que les parezca mejor. Sin embargo, desde el punto de vista del conductor, todas ellas funcionan del mismo modo. Esta misma idea se puede aplicar a la programación. El poder del código encapsulado es que cualquiera sabe cómo acceder al mismo y, por lo tanto, utilizarlo sin preocuparse de los detalles de la implementación, y sin temor a efectos inesperados.

En Java, la base de la encapsulación es la clase. Aunque examinaremos con más detalle las clases más adelante, una breve discusión sobre las mismas será útil ahora. Una *clase* define la estructura y comportamiento (datos y código) que serán compartidos por un conjunto de objetos. Cada objeto de una determinada clase contiene la estructura y comportamiento definidos por la clase, como si se hubieran grabado en ella con un molde con la forma de la clase. Por este motivo, algunas veces se hace referencia a los objetos como a *instancias de una clase*. Una clase es una construcción lógica, mientras que un objeto tiene una realidad física.

Cuando se crea una clase, se especifica el código y los datos que constituyen esa clase. En conjunto, estos elementos se denominan *miembros* de la clase. Específicamente, los datos definidos por la clase se denominan *variables miembro* o *variables de instancia*. Los códigos que operan sobre los datos se denominan *métodos miembro* o, simplemente, *métodos* (si está familiarizado con C o C++, en Java un programador denomina *método* a lo que en C/C++ un programador denomina *función*). En los programas correctamente escritos en Java, los métodos definen cómo se pueden utilizar las variables miembro. Esto significa que el comportamiento y la interfaz de una clase están definidos por los métodos que operan sobre sus datos de instancia.

Dado que el propósito de una clase es encapsular la complejidad, existen mecanismos para ocultar la complejidad de la implementación dentro de una clase. Cada método o variable dentro de una clase puede declararse como privada o pública. La interfaz *pública* de una clase representa todo lo que el usuario externo necesita o puede conocer. A los métodos y datos *privados* sólo se puede acceder por el código miembro de la clase. Por consiguiente, cualquier código que no sea miembro de la clase no tiene acceso a un método o variable privado. Puesto que los miembros privados de una clase sólo pueden ser accesados por otras partes del programa a través de los métodos públicos de la clase, eso asegura que no ocurran acciones impropias. Evidentemente, esto significa que la interfaz pública debe ser diseñada cuidadosamente para no exponer demasiado los trabajos internos de una clase (véase la Figura 2.1).

Herencia

La *herencia* es el proceso por el cual un objeto adquiere las propiedades de otro objeto. Esto es importante, ya que supone la base del concepto de clasificación jerárquica. Como se mencionó anteriormente, una gran parte del conocimiento se trata mediante clasificaciones jerárquicas. Por ejemplo, un labrador es parte de la clasificación de perros, que a su vez es parte de la clasificación de *mamíferos*, que está contenida en una clasificación mayor, la clase *animal*. Sin la utilización de jerarquías, cada objeto necesitaría definir explícitamente todas sus características. Sin embargo, mediante el uso de la herencia, un objeto sólo necesita definir aquellas cualidades que lo hacen único en su clase. Puede heredar sus atributos generales de sus padres. Por lo tanto, el mecanismo de la herencia hace posible que un objeto sea una instancia específica de un caso más general. Veamos este proceso con más detalle.

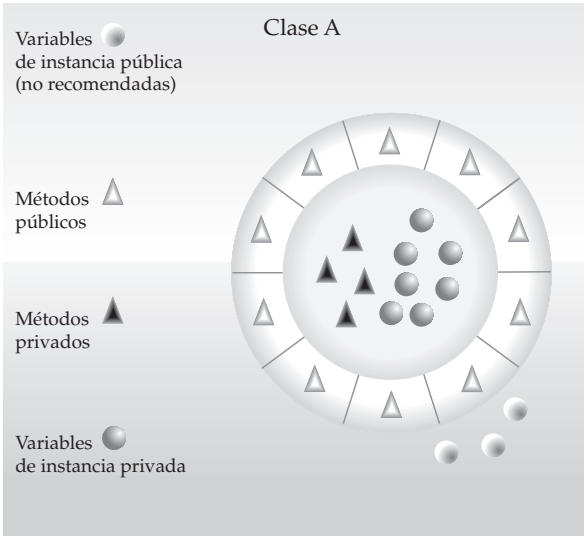
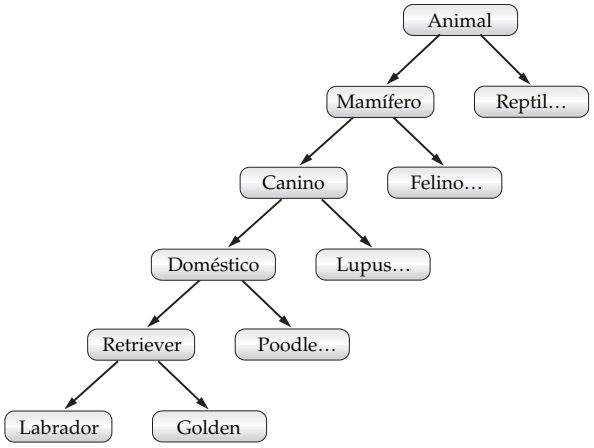


FIGURA 2.1
Encapsulación: se pueden utilizar métodos públicos para proteger datos privados.

Para muchas personas es natural considerar que el mundo está compuesto por objetos relacionados unos con otros de forma jerárquica, tal como los animales, los mamíferos y los perros. Si se quisiera describir a los animales de forma abstracta, se diría que tienen ciertos atributos, como tamaño, inteligencia y tipo de esqueleto. Los animales presentan también aspectos relativos al comportamiento, comen, respiran y duermen. Esta descripción de atributos y comportamiento es la definición de la *clase* de los animales.

Si se quisiera describir una clase más específica de animales, tales como los mamíferos, habría que indicar sus atributos específicos, como el tipo de dientes y las glándulas mamarias. A esto se denomina una *subclase* de animales, y la clase animal es una *superclase* de los mamíferos.

Como los mamíferos son simplemente unos animales especificados con más precisión, *heredan* todos los atributos de los animales. Una subclase hereda todos los atributos de cada uno de sus predecesores en la *jerarquía de clases*.



La herencia interactúa también con la encapsulación. Si una determinada clase encapsula determinados atributos, entonces cualquier subclase tendrá los mismos atributos *más* cualquiera que añada como parte de su especialización (véase la Figura 2.2). Éste es un concepto clave que permite a los programas orientados a objetos crecer en complejidad linealmente, en lugar de geométricamente. Una nueva subclase hereda todos los atributos de todos sus predecesores. Esto elimina interacciones impredecibles con gran parte del resto del código en el sistema.

Polimorfismo

El *polimorfismo* (del griego, "muchas formas") es una característica que permite que una interfaz sea utilizada por una clase general de acciones. La acción específica queda determinada por la

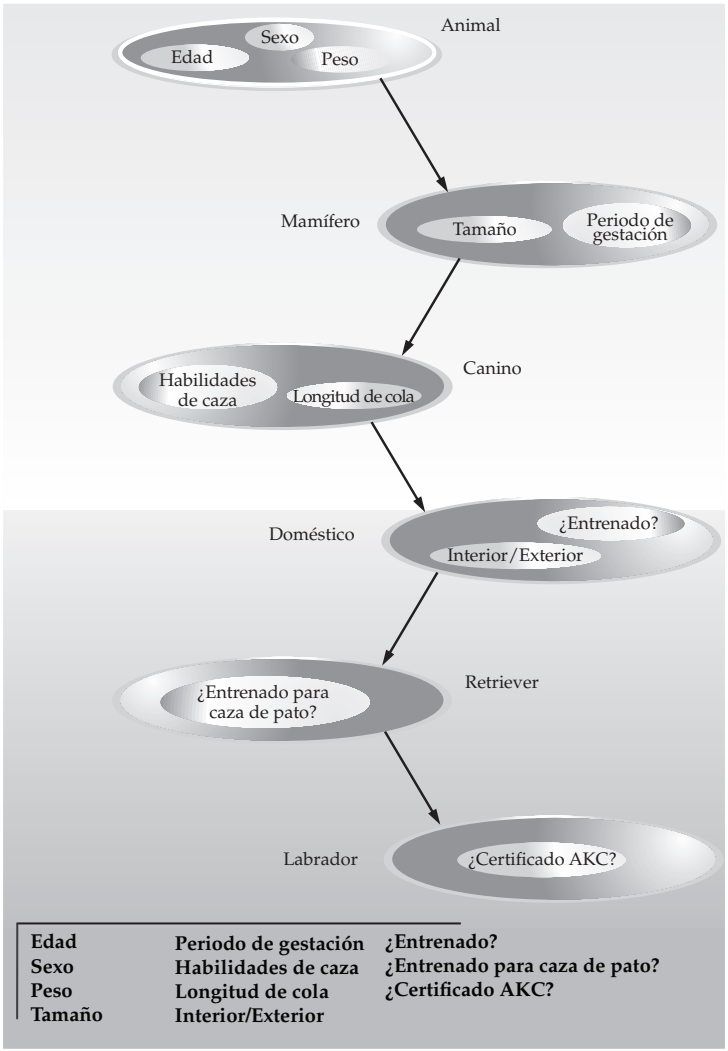


FIGURA 2.2. La clase Labrador hereda los elementos encapsulados de todas sus superclases.

naturaleza exacta de la situación. Consideremos una pila (que es una lista en la que el último elemento que entra es el primero que sale). Podríamos tener un programa que requiera tres tipos distintos de pilas. Una para valores enteros, otra para valores en punto flotante, y la última para caracteres. El algoritmo que implementa cada pila es el mismo, incluso aunque los datos almacenados sean diferentes. En un lenguaje no orientado a objetos sería necesario crear tres conjuntos diferentes de rutinas de pila, cada una con un nombre distinto. Sin embargo, gracias al polimorfismo, en Java se puede especificar un conjunto general de rutinas de pila que compartan los mismos nombres.

De manera más general, el concepto de polimorfismo se expresa a menudo mediante la frase "una interfaz, múltiples métodos". Esto significa que es posible diseñar una interfaz genérica para un grupo de actividades relacionadas. Esto ayuda a reducir la complejidad permitiendo que la misma interfaz sea utilizada para especificar una *clase general de acciones*. Es tarea del compilador seleccionar la *acción específica* (esto es, el método) que corresponde a cada situación. El programador no necesita hacer esta selección manualmente sólo recordar y utilizar la interfaz general.

Continuando con el ejemplo del perro, el sentido del olfato es polimórfico. Si el perro huele un gato, ladrará y correrá detrás de él. Si el perro huele comida, producirá saliva y correrá hacia su plato. El mismo sentido del olfato está funcionando en ambas situaciones. La diferencia está en lo que el perro huele, es decir, el tipo de dato sobre los que opera el olfato del perro. El mismo concepto general se implementa en Java cuando se aplican métodos dentro de un programa Java.

Polimorfismo, encapsulación y herencia trabajan juntos

Cuando se aplican adecuadamente, el polimorfismo, la encapsulación y la herencia dan lugar a un entorno de programación que facilita el desarrollo de programas más robustos y fáciles de ampliar que el modelo orientado a procesos. Una jerarquía de clase correctamente diseñada es la base que permite reutilizar un código en cuyo desarrollo y pruebas se han invertido tiempo y esfuerzo. La encapsulación permite trasladar las implementaciones en el tiempo sin tener que modificar el código que depende de las interfaces públicas de las clases. El polimorfismo permite crear un código claro, razonable, legible y elástico.

De los dos ejemplos del mundo real presentados, el del automóvil ilustra de forma más completa la potencia del diseño orientado a objetos. Es divertido pensar en los perros desde el punto de vista de la herencia, pero los coches se parecen más a los programas. Todos los conductores confían en la herencia para conducir diferentes tipos de vehículos (subclases). Tanto si el vehículo es un autobús escolar, un Mercedes sedán, un Porsche o un coche familiar, todos los conductores pueden encontrar y accionar más o menos el volante, los frenos o el acelerador. Después de cierta práctica con el mecanismo de cambio de velocidades, la mayoría de las personas pueden incluso superar la diferencia entre el cambio manual y el automático, ya que fundamentalmente conocen su superclase común, la transmisión.

Los conductores interactúan constantemente con características encapsuladas del automóvil. El freno y el acelerador son interfaces tremendamente sencillas sobre las que operan los pies, cuando se tiene en cuenta toda la complejidad que se esconde detrás de las mismas. La fabricación del motor, el estilo de los frenos y el tamaño de los neumáticos no tienen efecto alguno en la forma en que interactuamos con la definición de la clase de los pedales.

El último atributo, el polimorfismo, se refleja claramente en la capacidad de los fabricantes de coches de ofrecer una amplia gama de versiones del mismo vehículo básico. Por ejemplo, se puede elegir entre un coche con sistema de frenos ABS o con los frenos tradicionales, dirección asistida o normal y motor de 4, 6 u 8 cilindros. Cualquiera que sea el modelo elegido, habrá que presionar el

freno para parar, girar el volante para cambiar de dirección y presionar el acelerador para comenzar a moverse. La misma interfaz se puede utilizar para controlar distintas implementaciones.

Como se puede ver, mediante la aplicación del encapsulado, herencia y polimorfismo, las partes individuales se transforman en el objeto que conocemos como un coche. Lo mismo se puede decir de un programa de computadora. Por medio de la aplicación de los principios de la orientación a objetos, las diferentes partes de un programa complejo se unen para formar un todo cohesionado, robusto y sostenible.

Como se mencionó al comienzo de esta sección, todo programa en Java está orientado a objetos. O, de una forma más precisa, cada programa en Java implica encapsulación, herencia y polimorfismo. Aunque los ejemplos cortos que aparecen en el resto del capítulo y en los próximos capítulos puede que no exhiban claramente estas características, sin embargo, éstas están presentes. La mayor parte de las características de Java residen en sus bibliotecas de clases, que utilizan de forma amplia la encapsulación, la herencia y el polimorfismo.

Un primer programa sencillo

Una vez discutidos los pilares básicos de la orientación a objetos de Java, veamos programas de Java reales. Comencemos compilando y ejecutando el siguiente programa ejemplo, que, como se verá, supone algo más de trabajo de lo que se podría imaginar.

```
/*
    Este es un programa simple en Java.
    Este archivo se llama "Ejemplo.java".
*/
class Ejemplo {
  // El programa comienza con una llamada a main ().
  public static void main(String args []) {
    System.out.println("Este es un programa simple en Java.");
  }
}
```

NOTA *Las descripciones que siguen utilizan las herramientas de desarrollo de Java versión 6 de la empresa Sun Microsystems, (JDK 6, por sus siglas en inglés, Java Development Kit). Si se utiliza un entorno de desarrollo de Java diferente, puede ser necesario seguir un procedimiento distinto para compilar y ejecutar los programas de Java. En ese caso, habrá que consultar los manuales de usuario del compilador utilizado.*

Escribiendo el programa

En la mayor parte de los lenguajes de programación, el nombre del archivo que contiene el código fuente de un programa es irrelevante. Sin embargo, en Java esto no es así. La primera cuestión que hay que aprender en Java es que el nombre del archivo fuente es muy importante. Para este ejemplo, el nombre del archivo fuente debe ser **Ejemplo.java**. Veamos por qué.

En Java, un archivo fuente se denomina oficialmente *unidad de compilación*. Es un archivo de texto que contiene una o más definiciones de clase. El compilador Java requiere que el archivo fuente utilice la extensión **.java** en el nombre del archivo.

Volviendo al programa, el nombre de la clase definida por el programa es también **Ejemplo**. Esto no es una coincidencia. En Java, todo código debe residir dentro de una clase. Por convención, el nombre de esa clase debe ser el mismo que el del archivo que contiene el

programa. También es preciso asegurarse de que coinciden las letras mayúsculas y minúsculas del nombre del archivo y de la clase.

La razón es que Java distingue entre mayúsculas y minúsculas. En este momento, la convención de que los nombres de los archivos correspondan exactamente con los nombres de las clases puede parecer arbitraria. Sin embargo, esto facilita el mantenimiento y organización de los programas.

Compilando el programa

Para compilar el programa **Ejemplo**, se ejecuta el compilador, **javac**, especificando el nombre del archivo fuente en la línea de comandos, tal y como se muestra a continuación.

```
C:\>javac Ejemplo.java
```

El compilador **javac** crea un archivo llamado **Ejemplo.class**, que contiene la versión del programa en bytecode. Como se dijo anteriormente, el bytecode es la representación intermedia del programa que contiene las instrucciones que el intérprete o máquina virtual de Java ejecutará. Por tanto, el resultado de la compilación con javac no es un código que pueda ser directamente ejecutado.

Para ejecutar el programa realmente, se debe utilizar el intérprete denominado **java**. Para ello se pasa el nombre de la clase, **Ejemplo**, como argumento a la línea de comandos.

```
C:\>java Ejemplo
```

Cuando se ejecuta el programa, se despliega la siguiente salida:

```
Este es un programa simple en Java.
```

Cuando se compila código fuente, cada clase individual se almacena en su propio archivo, con el mismo nombre de la clase y utilizando la extensión .**class**. Ésta es la razón por la que conviene nombrar los archivos fuente con el mismo nombre que la clase que contienen, ya que así el nombre del archivo fuente coincidirá con el nombre del archivo .**class**. Cuando se ejecute el intérprete de **java**, se especificará realmente el nombre de la clase que se quiere que el intérprete ejecute. El intérprete automáticamente buscará un archivo con ese nombre y con la extensión **.class**. Si encuentra el archivo, ejecutará el código contenido en la clase especificada.

Análisis detallado del primer programa de prueba

Aunque **Ejemplo.java** es bastante corto, incluye varias de las características clave comunes a todos los programas en Java. Examinemos con más detalle cada parte del programa.

El programa comienza con las siguientes líneas:

```
/*
  Este es un programa simple en Java.
  Este archivo se llama "Ejemplo.java".
*/
```

Esto es un *comentario*. Como en la mayoría de los lenguajes de programación, Java permite introducir notas en el archivo fuente del programa. El contenido de un comentario es ignorado por el compilador. Un comentario describe o explica la operación del programa a cualquiera que esté leyendo el código fuente. En este caso, el comentario describe el programa y recuerda que el archivo fuente debe llamarse **Ejemplo.java**. Naturalmente, en aplicaciones reales, los comentarios generalmente explican cómo funcionan o qué hace alguna parte del programa.

Java proporciona tres tipos de comentarios. El que aparece al comienzo de este programa se denomina comentario multilínea. Este tipo de comentario debe comenzar con /* y terminar con */. Cualquier cosa que se encuentre entre los dos símbolos de comentario es ignorada por el compilador. Tal y como indica el nombre, un comentario multilínea puede tener varias líneas de longitud.

A continuación se muestra la siguiente línea de código del programa:

```
class Ejemplo {
```

Esta línea utiliza la palabra clave **class** para declarar que se está definiendo una nueva clase. **Ejemplo** es un *identificador* y el nombre de la clase. La definición completa de la clase, incluyendo todos sus miembros, debe estar entre la llave de apertura ({) y la de cierre (}). De momento, no nos preocuparemos más de los detalles de una clase, pero sí tendremos en cuenta que todas las acciones de un programa ocurren dentro de una clase. Ésta es una razón por la que todos los programas están orientados a objetos.

La siguiente línea del programa se muestra a continuación y es un *comentario de una línea*.

```
// El programa comienza con una llamada a main() .
```

Éste es el segundo tipo de comentarios que permite Java. Un comentario de *una sola línea* comienza con un // y termina al final de la línea. Como regla general, los programadores utilizan comentarios de múltiples líneas para notas más largas y comentarios de una sola línea para descripciones breves. El tercer tipo de comentario, el *comentario de documentación*, será analizado en la sección "Comentarios" más adelante en este capítulo.

A continuación se presenta la siguiente línea de código:

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

En esta línea comienza el método **main( )**. Tal y como sugiere el comentario anterior, en esta línea comienza la ejecución del programa. Todos los programas de Java comienzan la ejecución con la llamada al método **main( )**. El significado exacto de cada parte de esta línea no se puede precisar en este momento, ya que supone un conocimiento detallado del concepto de encapsulación en Java. Sin embargo, ya que en la mayoría de los ejemplos de la primera parte de este libro se usa esta línea de código, veamos brevemente cada parte de esta línea.

La palabra clave **public** es un *especificador de acceso* que permite al programador controlar la visibilidad de los miembros de una clase. Cuando un miembro de una clase va precedido por el especificador **public**, entonces es posible acceder a ese miembro desde cualquier código fuera de la clase en que se ha declarado (lo opuesto al especificador **public** es **private**, que impide el acceso a un miembro declarado como tal desde un código fuera de su clase). En este caso, **main( )** debe declararse como **public**, ya que debe ser llamado por un código que está fuera de su clase cuando el programa comienza. La palabra clave **static** (estático) permite que se llame a **main( )** sin tener que referirse a ninguna instancia particular de esa clase. Esto es necesario, ya que el intérprete o máquina virtual de Java llama a **main( )** antes de que se haya creado objeto alguno. La palabra clave **void** simplemente indica al compilador que **main( )** no devuelve ningún valor. Como se verá, los métodos pueden devolver valores. No se preocupe si todo esto resulta un tanto confuso. Todos estos conceptos se analizarán con más detalle en los capítulos siguientes.

Según lo indicado, **main( )** es el primer método al que se llama cuando comienza una aplicación Java. Hay que tener en cuenta que Java distingue entre mayúsculas y minúsculas, es decir, que **Main** es distinto de **main**. Es importante comprender que el compilador Java compilará

clases que no contengan un método **main()**, pero el intérprete de Java no puede ejecutar dichas clases. Así, si se escribe **Main** en lugar de **main**, el compilador compilará el programa, pero el intérprete de **java** enviará un mensaje de error al no poder encontrar el método **main()**.

Cualquier información que sea necesaria pasar a un método se almacena en las variables especificadas dentro de los paréntesis que siguen al nombre del método. A estas variables se las denomina *parámetros*. Aunque un determinado método no necesite parámetros, es necesario poner los paréntesis vacíos. En el método **main( )** sólo hay un parámetro, aunque complicado. **String args[ ]** declara un parámetro denominado **args**, que es un arreglo de instancias de la clase **String** (los *arreglos* son colecciones de objetos similares). Los objetos del tipo **String** almacenan cadenas de caracteres. En este caso, **args** recibe los argumentos que estén presentes en la línea de comandos cuando se ejecute el programa. Este programa no hace uso de esta información, pero otros programas que se presentan más adelante sí lo harán.

El último carácter de la línea es {. Este carácter señala el comienzo del cuerpo del método **main( )**. Todo el código comprendido en un método debe ir entre la llave de apertura del método y su correspondiente llave de cierre.

El método **main( )** es simplemente un lugar de inicio para el programa. Un programa complejo puede tener una gran cantidad de clases, pero sólo es necesario que una de ellas tenga el método **main( )** para que el programa comience. Cuando se comienza a crear applets -programas Java incrustados en navegadores Web- no se utilizará el método **main( )**, ya que el navegador Web utiliza un medio diferente para comenzar la ejecución de los applets.

A continuación se presenta la siguiente línea de código que está contenida dentro de **main( )**.

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

Esta línea despliega la cadena "Este es un programa simple en Java", seguida por una nueva línea en la pantalla. La salida es efectuada realmente por el método **println( )**. En este caso, el método **println( )** despliega la cadena de caracteres que se le pasan como parámetro. También se puede utilizar este método para visualizar información de otros tipos. La línea comienza con **System.out**. Aunque su explicación resulta complicada en este momento, se puede decir brevemente que **System** es una clase predefinida que proporciona acceso al sistema, y **out** es el flujo de salida que está conectado a la consola.

Como probablemente ya habrá adivinado, la salida y entrada por consola no se utilizan con frecuencia en los programas Java reales y applets. La mayor parte de las computadoras actuales tienen entonos gráficos con ventanas, por este motivo la E/S por consola solamente se utiliza en programas sencillos o de demostración. En capítulos posteriores se verán otras formas de generar salidas con Java, pero, de momento, continuaremos utilizando los métodos de E/S por consola.

Observe también que la sentencia **println( )** termina con un punto y coma. Todas las sentencias de Java terminan con un punto y coma. La razón para que otras líneas del programa no lo hagan así es que no son técnicamente sentencias.

La primera } del programa termina el método **main( )**, y la última } termina la definición de la clase **Ejemplo**.

Un segundo programa breve

Uno de los conceptos fundamentales en cualquier lenguaje de programación es el de variable. Una *variable* es un espacio de memoria con un nombre asignado, al que el programa puede asignar un valor. El valor de la variable se puede cambiar durante la ejecución del programa. El siguiente programa muestra cómo se declara una variable y cómo se le asigna un valor. Además,

también ilustra algunos aspectos nuevos de la salida por consola. Como indican los comentarios de las primeras líneas, el archivo correspondiente debe llamarse **Ejemplo2.java**.

```
/*
    Este es otro ejemplo breve.
    Este archivo se llama "Ejemplo2.java".
*/
class Ejemplo2 {
  public static void main(String args[]) {
    int num; // declara una variable llamada num

    num = 100; // asigna a num el valor 100

    System.out.println("Este es num: " + num);

    num = num * 2;

    System.out.print{"El valor de num * 2 es ");
    System.out.println(num);
  }
}
```

Al ejecutar este programa, se obtiene la siguiente salida:

```
Este es num: 100
El valor de num * 2 es 200
```

Veamos con más detalle cómo se produce esta salida. La primera línea nueva del programa es:

```
int num; // declara una variable llamada num
```

Esta línea declara una variable entera llamada **num**. Como muchos otros lenguajes, Java requiere que las variables sean declaradas antes de utilizarlas.

La forma general de declaración de una variable es:

tipo nombre;

Donde *tipo* especifica el tipo de la variable declarada, y *nombre* es el nombre de la variable. Se puede declarar más de una variable de un tipo determinado separando por comas los nombres de las variables a declarar. Java define varios tipos de datos entre los que se pueden citar los enteros, caracteres y punto flotante. La palabra clave **int** especifica un tipo entero.

En el programa, la línea

```
num = 100; // asigna a num el valor 100
```

asigna a **num** el valor 100. En Java, el operador de asignación es el signo igual. La siguiente línea del código es la responsable de desplegar el valor de **num** precedido por la cadena de caracteres "Esto es num:".

```
System.out.println("Este es num: " + num);
```

En esta sentencia, el signo de suma hace que el valor de **num** sea añadido a la cadena que le precede, y a continuación se despliega la cadena resultante. Lo que realmente ocurre es que **num** se convierte en el carácter equivalente y después se concatena con la cadena que le precede. Este proceso se describirá con más detalle más adelante. Este mecanismo se puede generalizar. Utilizando el operador +, se pueden encadenar tantos elementos como se desee dentro de una única sentencia **println**( ).

La siguiente línea de código asigna a **num** el valor de **num** multiplicado por dos. Como en otros lenguajes, Java utiliza el operador * para indicar multiplicación. Después de la ejecución de esta línea, el valor almacenado en **num** será 200.

Las dos siguientes líneas de programa son:

```
System.out.print ("El valor de num * 2 es ");
System.out.println(num);
```

En estas dos líneas hay cosas que aparecen por primera vez. En primer lugar, el método **print( )** se utiliza para presentar la cadena"El valor de num * 2 es". Esta cadena no es seguida por una nueva línea. Esto significa que cuando se genere una nueva salida, comenzará en la misma línea. El método **print( )** es como el método **println( )**, excepto que no pone el carácter de línea nueva después de cada llamada. A continuación, en la llamada a **println( )** se utiliza **num** para imprimir el valor almacenado en la variable. Para la salida de valores de cualquier tipo en Java se pueden utilizar ambos métodos, **print( )** y **println( )**.

Dos sentencias de control

Aunque en el Capítulo 5 se examinan con más profundidad las sentencias de control, a continuación se introducen brevemente dos de ellas para que se puedan utilizar en los programas de ejemplo que aparecen en los capítulos 3 y 4. Además nos servirán para explicar un importante aspecto de Java: los bloques de código.

La sentencia if

La sentencia **if** de Java actúa de la misma forma que la sentencia IF en cualquier otro lenguaje. Además, es sintácticamente idéntica a las sentencias **if** de C, C++ y C#. A continuación se presenta su forma más simple.

if*(condición) sentencia;*

donde *condición* es una expresión booleana. Si la *condición* es verdadera, entonces se ejecuta la sentencia. Si la *condición* es falsa, entonces se evita la sentencia. A continuación se presenta un ejemplo:

```
if(num < 100) println("num es menor que 100");
```

En este caso, si **num** contiene un valor menor que 100, la expresión condicional es verdadera, y se ejecutará la sentencia **println( )**. Si **num** contiene un valor mayor o igual que 100, entonces no se ejecuta el método **println( ).**

Como se verá en el Capítulo 4, Java define un conjunto completo de operadores relacionales que se pueden utilizar en expresiones condicionales. Algunos de éstos son:

Operador	Significado
<	Menor que
>	Mayor que
==	Igual a

Observe que para la prueba de igualdad se utiliza el doble signo igual.

El siguiente programa ejemplifica el uso de la sentencia **if**:

```
/*
  Demostración de la sentencia if.

  Este archivo se llama "EjemploIf.java".
*/
class EjemploIf {
  public static void main(String args[]) {
    int x, y;

    x = 10;
    y = 20;

    if(x < y) System.out.println ("x es menor que y");

    x = x * 2;
    if(x == y) System.out.println("x es ahora igual que y");

    x = x * 2;
    if(x > y) System.out.println ("x es ahora mayor que y");

    // Esto no desplegará nada
    if(x == y) System.out.println ("esto no se verá");
  }
}
```

La salida generada por este programa es la siguiente:

```
x es menor que y
x es ahora igual que y
x es ahora mayor que y
```

Observe otra cosa en este programa. La línea

```
int x, y;
```

declara dos variables, **x** e **y**, utilizando una lista con elementos separados por comas.

El ciclo for

Como es de sobra conocido, las sentencias de ciclos son una parte importante de prácticamente cualquier lenguaje de programación, y Java no es una excepción. De hecho, tal y como se verá en el Capítulo 5, Java facilita un potente surtido de construcciones de ciclos. Probablemente la más versátil es el ciclo **for**. La forma más simple del ciclo **for** es la siguiente:

for*(inicialización; condición; iteración) sentencia;*

En su forma más habitual, la parte de inicialización del ciclo asigna un valor inicial a la variable de control del ciclo. La condición es una expresión booleana que examina la variable de control del ciclo. Si el resultado de la prueba es verdadero, el ciclo **for** continúa iterando. Si es falso, el ciclo termina. La expresión de *iteración* determina cómo cambia la variable de control cada vez que se recorre el ciclo. El siguiente programa sirve como ejemplo del ciclo **for**:

```
/*
  Demostración del ciclo for.
```

```
  Este archivo se llama "ForPrueba.java".
*/
class ForPrueba {
  public static void main(String args[]) {
    int x;

    for(x = 0; x<l0; x = x+l)
      System.out.println ("Esta es x: "+ x);
  }
}
```

Este programa genera la siguiente salida:

```
Esta es x: 0
Esta es x: 1
Esta es x: 2
Esta es x: 3
Esta es x: 4
Esta es x: 5
Esta es x: 6
Esta es x: 7
Esta es x: 8
Esta es x: 9
```

En este ejemplo, la variable de control del ciclo es **x**. En la parte de inicialización del ciclo **for**, esta variable se inicializa a cero. Al comienzo de cada iteración (incluyendo la primera) se comprueba la condición x < 10. Si el resultado es verdadero, se ejecuta la sentencia **println( )**, y a continuación se ejecuta la parte de la iteración del ciclo. Este proceso continúa hasta que la condición sea falsa.

En los programas profesionales escritos en Java, casi nunca se verá la parte de iteración del ciclo escrita tal y como se ha hecho en el programa anterior. Es decir, raramente se verá una sentencia como ésta:

```
x = x + 1;
```

La razón es que Java incluye un operador especial de incremento que realiza esta operación de forma más eficiente. El operador de incremento es **++**, es decir, dos signos de suma consecutivos. El operador de incremento incrementa su operando en una unidad. Utilizando este operador, la sentencia anterior se escribe de la siguiente forma:

```
x++;
```

Por lo tanto, el ciclo **for** del programa anterior se escribirá normalmente así:

```
for(x = 0; x<l0; x++)
```

La ejecución de este ciclo produce exactamente la misma salida que el anterior.

Java también proporciona un operador de decremento, que se especifica como `--`. Este operador reduce su operando en una unidad.

Utilizando bloques de código

Java permite la agrupación de dos o más sentencias en los denominados bloques de código. Para ello se encierran entre llaves las sentencias del bloque. Una vez creado, un bloque de código se convierte en una unidad lógica que se puede utilizar en cualquier sitio de la misma forma que una sentencia única. Por ejemplo, un bloque de código puede ser el cuerpo de las sentencias **if** o **for** de Java. Consideremos esta sentencia **if**:

```
if(x < y) { // comienzo del bloque
  x = y;
  y = 0;
} // fin del bloque
```

Donde si **x** es menor que **y**, entonces las dos sentencias que están dentro del bloque se ejecutan. Por lo tanto, las dos sentencias del bloque forman una unidad lógica, y una sentencia no puede ejecutarse sin que también se ejecute la otra. Siempre que se necesite unir lógicamente dos o más sentencias, esto se consigue creando un bloque.

Veamos otro ejemplo. El siguiente programa utiliza un bloque de código como cuerpo de un ciclo **for**.

```
/*
  Demostración de un bloque de código.

  Este archivo se llama "PruebaBloque.java"
*/
class PruebaBloque {
  public static void main(String args[]) {
    int x, y;

    y = 20;

    // el cuerpo de este ciclo es un bloque
    for(x = 0; x<10; x++} {
      System.out.println ("Esta es x: " + x);
      System.out.println("Esta es y: " + y);
      y = y - 2;
    }
  }
}
```

La salida generada por este programa es la siguiente:

```
Esta es x: 0
Esta es y: 20
Esta es x: 1
Esta es y: 18
Esta es x: 2
Esta es y: 16
Esta es x: 3
Esta es y: 14
Esta es x: 4
```

```
Esta es y: 12
Esta es x: 5
Esta es y: 10
Esta es x: 6
Esta es y: 8
Esta es x: 7
Esta es y: 6
Esta es x: 8
Esta es y: 4
Esta es x: 9
Esta es y: 2
```

En este caso, el cuerpo del ciclo **for** es un bloque y no una única sentencia. Por lo tanto, cada vez que se realiza una iteración, se ejecutan las tres sentencias que están dentro del bloque, tal y como muestra la salida generada por el programa.

Como se verá más adelante en este libro, los bloques de código tienen propiedades y usos adicionales. Sin embargo, la principal razón para su existencia es crear unidades lógicas de código inseparables.

Cuestiones de léxico

Ahora que se han presentado varios programas pequeños, podemos describir de manera más formal los elementos básicos de los programas escritos en Java. Los programas en Java son una colección de espacios en blanco, identificadores, literales, comentarios, operadores, separadores y palabras clave. Los operadores se describen en el próximo capítulo. Los demás se describen a continuación.

Espacios en blanco

En Java no es necesario seguir reglas especiales de indentación. Por ejemplo, el programa **Ejemplo** se podría haber escrito en una línea o de cualquier otra extraña forma en que se hubiese deseado, siempre y cuando hubiera un carácter de espacio en blanco entre cada elemento que no estuviera ya delimitado por un operador o separador. En Java un espacio, tabulador o línea nueva son un espacio en blanco.

Identificadores

Los identificadores se utilizan para los nombres de las clases, de los métodos y de las variables. Un identificador puede ser cualquier secuencia descriptiva de letras mayúsculas o minúsculas, números, el carácter de subrayado, o el símbolo del dólar. Un identificador no debe empezar nunca con un número, para evitar la confusión con un literal numérico. Conviene recordar otra vez que Java distingue entre mayúsculas y minúsculas; así, el identificador **VALOR** no es lo mismo que el identificador **valor**. Éstos son algunos ejemplos de identificadores válidos:

TempMedia	cuenta	a4	$prueba	esto_es_correcto

Como ejemplos de identificadores no válidos tenemos:

2cuenta	Temp-alta	No/correcto

Literales

Un valor constante en Java se crea mediante una representación *literal*. Por ejemplo,

100	98.6	'X'	"Esto es una prueba"

De izquierda a derecha, el primer literal especifica un entero; el segundo, un valor en punto flotante; el tercero, un carácter constante, y el último, una cadena de caracteres. Se puede utilizar una literal en cualquier parte en la que un valor de este tipo esté permitido.

Comentarios

Existen tres tipos de comentarios definidos por Java. Ya se han visto los dos primeros: el de una sola línea y el de múltiples líneas. El tercer tipo de comentario es el denominado *comentario de documentación*. Este tipo de comentario se utiliza para generar un archivo HTML que documente el programa. El comentario de documentación comienza con un /** y termina con un */. En el Apéndice A se explican los comentarios de documentación.

Separadores

En Java, se utilizan unos pocos caracteres como separadores. El más utilizado es el punto y coma. Los separadores se utilizan para indicar el final de una sentencia. La siguiente tabla muestra los separadores válidos en Java:

Símbolo	Nombre	Propósito
( )	Paréntesis	Se usa para contener una lista de parámetros en la definición y llamada a un método. También se utilizan para definir la precedencia, contener expresiones en sentencias de control de flujo y en conversiones de tipo.
{ }	Llaves	Se usan para contener los valores de arreglos inicializados automáticamente. También para definir un bloque de código, para clases métodos y ámbitos locales.
[ ]	Corchetes	Se usan para declarar arreglos. También cuando se accede a valores contenidos en arreglos.
;	Punto y coma	Separador de sentencias.
,	Coma	Separa identificadores consecutivos en la declaración de variables. También se usa para encadenar sentencias dentro de un ciclo **for**.
.	Punto	Se usa para separar nombres de paquetes de nombres de subpaquetes y clases. También para separar una variable o método de una variable de referencia.

Palabras clave de Java

Existen actualmente 50 palabras clave reservadas, definidas en el lenguaje Java (véase la Tabla 2.1). Estas palabras clave, combinadas con la sintaxis de los operadores y separadores, forman la definición del lenguaje Java, y no se pueden utilizar como nombres de una variable, clase o método.

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

Tabla 2.1. Palabras reservadas de Java

Las palabras clave **const** y **goto** están reservadas pero no se usan. En los comienzos de Java existían muchas otras palabras clave reservadas para un posible uso futuro. Sin embargo, la especificación actual de Java sólo define las palabras clave que se muestran en la Tabla 2.1.

Además de las palabras clave, Java reserva las siguientes: **true**, **false** y **null**. Éstos son valores definidos por Java. No se pueden utilizar estas palabras como nombres de variables, clases, etcétera.

La biblioteca de clases de Java

Los ejemplos de programas mostrados en este capítulo utilizan dos métodos internos de Java: **println( )** y **print( )**. Estos métodos son miembros de la clase **System**, que es una clase predefinida por Java e incluida automáticamente en los programas. Dentro de una visión más amplia, el entorno Java se basa en varias bibliotecas de clases que contienen los métodos necesarios para acciones tales como E/S, gestión de cadenas de caracteres, redes y gráficos. Las clases estándares también proporcionan soporte para la salida en un entorno de ventanas. Por lo tanto, Java, en su totalidad, es una combinación del propio lenguaje Java y sus clases estándares. Las bibliotecas de clases facilitan mucha de la funcionalidad de Java. De hecho para llegar a ser un programador de Java es importante aprender a utilizar las clases estándares de Java. A lo largo de la Parte 1 de este libro, se describen elementos de la biblioteca de clases estándares de Java conforme se vayan necesitando. En la Parte II, se describen en detalle las bibliotecas de clases.

CAPÍTULO 3

Tipos de dato, variables y arreglos

Este capítulo examina tres de los elementos fundamentales de Java: tipos de datos, variables y arreglos. Como todos los lenguajes modernos de programación, Java soporta diversos tipos de datos. Se pueden utilizar estos tipos de datos para declarar variables y crear arreglos. Como se verá más adelante la propuesta de Java para variables y arreglos es clara, eficiente y consistente.

Java es un lenguaje fuertemente tipificado

Es importante establecer desde el principio que Java es un lenguaje fuertemente tipificado. Efectivamente, parte de la seguridad y robustez de Java se debe a este hecho. Veamos lo que esto significa. En primer lugar, cada variable y cada expresión tienen un tipo, y cada tipo está definido estrictamente. En segundo lugar, en todas las asignaciones, ya sean explícitas o por medio del paso de parámetros en la llamada a un método, se comprueba la compatibilidad de tipos. En Java no existen conversiones automáticas de tipos incompatibles como en algunos otros lenguajes. El compilador de Java revisa todas las expresiones y parámetros para asegurarse de que los tipos son compatibles. Cualquier incompatibilidad de tipos da lugar a errores, que deben ser corregidos antes de que el compilador termine de compilar la clase

Los tipos primitivos

Java define ocho tipos *primitivos*: **byte, short, int, long, char, float, double** y **boolean**. Los tipos primitivos son llamados también tipos *simples*, ambos términos serán utilizados en este libro. Estos tipos pueden ser clasificados en cuatro grupos:

- Enteros: este grupo incluye a los tipos **byte, short, int** y **long** para almacenar números enteros positivos y negativos.
- Números con punto decimal: este grupo incluye a los tipos **float** y **double**, los cuales representan números con precisión fraccional.
- Caracteres: este grupo incluye al tipo **char**, el cual representa símbolos en un conjunto de caracteres, como letras y números.
- Booleano: Este grupo incluye **boolean**, el cual es un tipo especial para representar valores lógicos (verdadero y falso).

Estos tipos se pueden utilizar por sí solos, o para construir arreglos o clases propias. Forman, por lo tanto, la base para todos los demás tipos de datos que se puedan crear.

Los tipos primitivos representan valores simples, no objetos complejos. Aunque Java es un lenguaje completamente orientado a objetos, los tipos primitivos no son objetos. Los tipos simples son análogos a los tipos simples existentes en la mayoría de los lenguajes no orientados a objetos. La razón de esto es la eficiencia. Haber definido los tipos simples como objetos habría reducido considerablemente la eficiencia.

Los tipos primitivos se definen de forma que tienen un rango y un comportamiento matemático explícito. Lenguajes como C y C++ permiten que el tamaño de un entero varíe según lo establezca el entorno de ejecución. Sin embargo, Java es diferente. Debido al requerimiento de portabilidad, todos los tipos de datos tienen un rango estrictamente definido. Por ejemplo, un int tiene siempre 32 bits, independientemente de la plataforma. Esto permite garantizar que los programas escritos se podrán ejecutar *sin problema* en cualquier máquina sin realizar modificación alguna. Aunque especificar estrictamente el tamaño de un entero puede causar pequeñas disminuciones del rendimiento en algunos ambientes, es necesario hacerlo para garantizar portabilidad.

Veamos cada tipo de dato.

Enteros

Java define cuatro tipos de enteros: **byte, short, int** y **long**. En todos ellos se considera el signo, valores positivos y negativos. Java no admite valores sin signo. Muchos otros lenguajes soportan enteros con signo y enteros sin signo. Sin embargo, los diseñadores del Java creyeron que los enteros sin signo eran innecesarios. Específicamente, creyeron que el concepto de sin signo se utilizaba en la mayoría de los casos para especificar el comportamiento del *bit más significativo* que definía el *signo* del valor numérico. Como se verá en el Capítulo 4, Java trata de forma diferente el significado del bit más significativo, añadiendo un operador especial de desplazamiento a la derecha sin signo. Con este operador, se eliminó la necesidad de un tipo entero sin signo.

No se debe pensar en el *tamaño* de un entero como la cantidad de memoria que requiere, sino más bien como el *comportamiento* que implica en variables y expresiones de ese tipo. El intérprete de Java puede utilizar el tamaño que quiera, en tanto los tipos se comporten acorde a lo esperado por los tipos que se han declarado. El tamaño y rango de estos tipos enteros puede variar mucho, según se muestra en la tabla siguiente:

Nombre	Tamaño	Rango
long	64	–9,223,372,036,854,775,808 a 9,223,372,854,775,807
int	32	–2,147,483,648 a 2,147,483,647
short	16	–32,768 a 32,767
byte	8	–128 a 127

Veamos cada uno de los tipos enteros

byte

El tipo entero más pequeño es el **byte**. Este es un tipo de 8 bits con signo que tiene un rango desde –128 a 127. Las variables del tipo **byte** son especialmente útiles cuando se está trabajando

con un flujo de datos que procede de la red o de un archivo. También son útiles cuando se está trabajando con datos binarios que pueden no ser directamente compatibles con otros tipos de Java.

Las variables del tipo **byte** se declaran mediante la palabra clave **byte**. Como ejemplo, se declaran a continuación dos variables llamadas **b** y **c** de tipo **byte**.

```
byte b, c;
```

short

short es un tipo de 16 bits con signo. Este tipo tiene un rango que va desde –32,768 a 32,767. Es probablemente el tipo menos utilizado en Java. Aquí tenemos dos ejemplos de la declaración de variables **short**:

```
short s;
short t;
```

int

El tipo entero más utilizado es **int**. Es un tipo de 32 bits con signo que tiene un rango de –2,147,483,648 a 2,147,483,647. Además de otros usos, las variables del tipo **int** se emplean normalmente para el control de ciclos y como índices de arreglos. Aunque se podría pensar que utilizar una variable de tipo **byte** o **short** podría ser más eficiente que utilizar un tipo **int** en situaciones en las cuales un rango grande como el del **int** no es necesario, esto no es el caso. La razón es que cuando valores del tipo **byte** o **short** son utilizados en una expresión, estos valores son *convertidos* en **int** en el momento en que la expresión es evaluada (la conversión de tipos se describe más adelante en este capítulo). Por esta razón, el tipo **int** es frecuentemente la mejor opción para trabajar con valores enteros.

long

long es un tipo de 64 bits con signo y es adecuado para aquellas ocasiones en las que el tipo **int** no es lo suficientemente grande para almacenar un determinado valor. El rango de **long** es amplio. Esto hace que este tipo sea útil cuando se necesita trabajar con números muy grandes. Como ejemplo, en el siguiente programa se calcula el número de millas que recorre la luz en un número de días especificado.

```
// Cálculo de la distancia que recorre la luz usando variables long
class Luz {
  public static void main (String args[]){
    int velocidad;
    int dias;
    int segundos;
    long distancia;

    //velocidad aproximada de la luz en millas por segundo
    velocidad = 186000;

    dias = 1000; //aquí se especifica el número de días
    segundos = dias * 24 * 60 * 60; // conversión a segundos

    distancia = velocidad * segundos; // cálculo de la distancia

    System.out.print ( "En" + días);
```

```
    System.out.print ( "días la luz recorrerá aproximadamente ");
    System.out.print ( distancia + " millas");
  }
}
```

Este programa genera la siguiente salida:

```
En 1000 días la luz recorrerá aproximadamente 16070400000000 millas
```

Es evidente que el resultado no podría haber sido almacenado en una variable **int**.

Tipos con punto decimal

Los números con punto decimal, también conocidos como números reales, se utilizan para evaluar expresiones que requieren precisión decimal. Por ejemplo, cálculos como el de una raíz cuadrada, o los de las funciones trascendentes como seno y coseno, dan lugar a valores cuya precisión requiere el tipo de punto decimal. Java implementa el conjunto estándar (IEEE-754) de tipos y operadores en punto decimal. Existen dos clases de tipos con punto decimal, **float** y **double**, que representan números con precisión simple y doble, respectivamente. Su tamaño y rango se muestran a continuación

Nombre	Tamaño	Rango aproximado
double	64	4.9e–324 a 1.8e+308
float	32	1.4e–045 a 3.4e+038

A continuación se analiza cada uno de estos dos tipos.

float

El tipo **float** especifica un valor en *precisión simple* que utiliza 32 bits. Las operaciones en precisión simple son más rápidas y utilizan la mitad de espacio de memoria que en precisión doble, pero son poco precisas si los valores resultantes son muy grandes o muy pequeños. Las variables del tipo **float** son útiles cuando se necesita un valor fraccionario pero no se requiere un alto grado de precisión. Por ejemplo, se puede utilizar **float** para representar dólares y centavos.

Éstos son unos ejemplos de declaración de variables **float**:

```
float tempmax, tempmin;
```

double

Los valores en doble precisión se denotan mediante la palabra clave **double**, y se utilizan 64 bits para almacenar un valor. En algunos procesadores modernos las operaciones en precisión doble son más rápidas que en precisión simple, ya que éstos han sido optimizados para obtener una mayor velocidad en cálculos matemáticos. Todas las funciones matemáticas trascendentales, como las funciones **sin()**, **cos()** y **sqrt()**, devuelven valores del tipo **double**. El tipo **double** es la mejor elección cuando se necesita mantener la exactitud a lo largo de varios cálculos iterativos, o se está trabajando con números muy grandes.

En el siguiente programa se utilizan variables del tipo **double** para calcular el área de un círculo.

```
// Cálculo del área de un círculo.
class Area {
  public static void main(String args[]) {
    double pi, r, a;

    r = 10.8; // radio del círculo
    pi = 3.1416; // valor aproximado de pi
    a = pi * r * r; // cálculo del área

    System.out.println("El área del círculo es " + a);
  }
}
```

Caracteres

El tipo de datos que se utiliza en Java para almacenar caracteres es **char**. Sin embargo, los programadores de C/C++ deben tener cuidado: el tipo **char** de Java no es lo mismo que el tipo **char** de C o C++. En C/C++, char es un tipo entero de 8 bits. Esto no es el caso de Java, ya que este lenguaje utiliza Unicode para representar caracteres. *Unicode* define un conjunto completo e internacional de caracteres que permite la representación de todos los caracteres que se pueden encontrar en todas las lenguas de la humanidad. Unicode es una unificación de un gran número de conjuntos de caracteres, tales como los del latín, griego, árabe, cirílico, hebreo, katakana, hangul y muchos más. Para ello son necesarios 16 bits. Por este motivo, el tipo **char** de Java es un tipo de 16 bits. El rango de un **char** es de 0 a 65,536. No existen valores de tipo **char** negativos. El conjunto estándar de caracteres conocido como ASCII tiene un rango que va de 0 a 127 caracteres, y el conjunto extendido de 8 bits, ISOLatin-l, va desde 0 a 255. Parece lógico que Java utilice el conjunto Unicode para representar caracteres, ya que está diseñado para crear aplicaciones que puedan ser utilizadas en todo el mundo. Naturalmente, la utilización de Unicode puede resultar ineficiente para lenguas como el inglés, alemán, español o francés, cuyo conjunto de caracteres se puede representar fácilmente con 8 bits. Pero éste es el precio que hay que pagar para conseguir la portabilidad global.

NOTA *Se puede obtener más información sobre Unicode en http://www.unicode.org.*

El siguiente programa utiliza variables del tipo **char**:

```
// Ejemplo de datos del tipo char.
class CharDemo {
  public static void main(String args[]) {
    char ch1, ch2;

    ch1 = 88; // codificación de X
    ch2 = 'Y';

    System.out.print("ch1 y ch2: ");
    System.out.println(ch1 + " " + ch2);
  }
}
```

Este programa da lugar a la siguiente salida:

```
ch1 y ch2: X Y
```

Observe que a **ch1** se le asigna el valor 88, que es el valor ASCII (y Unicode) correspondiente a la letra X. Como se ha mencionado, el conjunto de caracteres ASCII ocupa los primeros 127 valores en el conjunto de caracteres Unicode. Por esta razón todos los "viejos trucos" utilizados con caracteres en otros lenguajes también sirven en Java.

Aunque los datos del tipo **char** no son enteros, en muchos casos se puede operar con ellos como si lo fueran. Por ejemplo, se puede sumar dos caracteres o incrementar el valor de una variable de este tipo. Consideremos el siguiente programa:

```
// Las variables char se comportan como enteros
class CharDemo2 {
  public static void main (String args[]) {
    char ch1;

    ch1 = 'X';
    System.out.println ("ch1 contiene " + ch1 );

    ch1 ++; // incremento de ch1
    System.out.println ("ch1 es ahora " + ch1);
  }
}
```

La salida generada por este programa es la que se muestra a continuación:

```
ch1 contiene X
ch1 es ahora Y
```

En el programa se asigna el valor X a **ch1** en primer lugar. A continuación se incrementa **ch1**. El resultado es que entonces **ch1** contiene el valor Y, que es el siguiente carácter en la secuencia ASCII (y Unicode).

Booleanos

Java tiene un tipo primitivo, denominado **boolean**, para valores lógicos. Una variable de este tipo sólo puede tener dos valores, **true** o **false**. Éste es el tipo que devuelven los operadores relacionales tales como **a < b. boolean** es también el tipo requerido por las expresiones condicionales que gobiernan las sentencias de control, como **if** y **for**.

El siguiente programa es un ejemplo del tipo **boolean**:

```
// Ejemplo de valores boolean
class BoolTest {
  public static void main (String args[]) {
    boolean b;

    b = false;
    System.out.println ("b es " + b);
    b = true;
    System.out.println ("b es " + b);

    // un valor booleano puede controlar una sentencia if

    if (b) System.out.println ("Esto se ejecuta");

    b = false;
    if (b) System.out.println ("Esto no se ejecuta");

    // El resultado de una operación relacional es un valor booleano
```

```
    System.out.println ("10 > 9 es " + (10 > 9) );
  }
}
```

La salida generada por este programa es la siguiente:

```
b es false
b es true
Esto se ejecuta
10 > 9 es true
```

En este programa vale la pena observar tres cosas. En primer lugar, como se puede ver, cuando un valor **boolean** es presentado por **println()**, lo que se imprime es"true"o"false". En segundo lugar, el valor de una variable del tipo **boolean** es suficiente por sí mismo, para el control de la sentencia **if**. No es necesario escribir una sentencia **if** como la siguiente:

```
if ( b == true) …
```

En tercer lugar, el resultado de un operador relacional, tal como **<**, es un valor del tipo **boolean**. Ésta es la razón de que la expresión **10 > 9** muestre el valor "true". El conjunto de paréntesis que encierran a **10 > 9** es necesario porque el operador **+** tiene prioridad sobre el operador **>**.

Una revisión detallada de los valores literales

En el Capítulo 2 se mencionaron brevemente los literales. Veámoslos con más detalle ahora después de haber descrito los tipos de Java.

Literales enteros

El tipo más utilizado en los programas de Java es probablemente el de los enteros. Cualquier valor numérico entero es un literal entero, como por ejemplo, 1, 2, 3 y 42. Éstos son valores decimales, es decir, escritos en base 10. Existen otras dos bases que se pueden utilizar para literales enteros, la octal (base 8) y la hexadecimal (base 16). En Java se indica que un valor es octal porque va precedido por un 0. Por lo tanto, el valor aparentemente válido 09 producirá un error de compilación, ya que 9 no pertenece al conjunto de dígitos utilizados en base 8 que van de 0 a 7. Una base más utilizada por los programadores es la hexadecimal, que corresponde claramente con las palabras de tamaño de módulo 8 tales como las de 8, 16, 32 y 64 bits. Una constante hexadecimal se denota precediéndola por un cero-x (**0x** o **0X**). Los dígitos que se utilizan en base hexadecimal son del 0 al 9, y las letras de la *A* a la *F* (o de la a a la *f*), que sustituyen a los números del 10 al 15.

Los literales enteros crean un valor **int**, que es un valor entero de 32 bits. Teniendo en cuenta que Java es un lenguaje fuertemente tipificado, nos podríamos preguntar cómo es posible asignar un literal entero a alguno de los otros tipos de enteros de Java **byte** o **long**, sin que se produzca un error de incompatibilidad. Afortunadamente, estas situaciones se resuelven de forma sencilla. Cuando se asigna una literal a una variable del tipo **byte** o **short**, no se genera ningún error si el valor literal está dentro del rango del tipo de la variable.

Siempre es posible asignar un literal entero a una variable de tipo **long**. Sin embargo, para especificar un literal de tipo **long** es preciso indicar de manera explícita a la computadora que el valor literal es del tipo **long**. Esto se hace añadiendo la letra L mayúscula o minúscula al literal. Por ejemplo, 0x7fffffffffffffL o 9223372036854775807 es el mayor literal del tipo **long**. Un entero

puede ser asignado también a una variable de tipo **char** mientras se encuentre dentro del rango establecido para **char**.

Literales con punto decimal

Los números con punto decimal representan valores decimales con un componente fraccional. Se pueden representar utilizando las notaciones estándar o científica. La *notación estándar* consiste en la parte entera, el punto decimal y la parte fraccional. Por ejemplo, 2.0, 3.14159 y 0.6667 son representaciones válidas en la notación estándar de números de punto flotante. La *notación científica* utiliza además de la notación estándar un sufijo que especifica la potencia de 10 por la que hay que multiplicar el número. El exponente se indica mediante una *E* o *e* seguida de un número decimal, que puede ser positivo o negativo; por ejemplo, 6.022E23, 3.14159E-05, y 2E+100.

Los literales de punto flotante, en Java, utilizan por omisión la precisión **double**. Para especificar un literal de tipo **float** se debe añadir una *F* o *f* a la constante. También se puede especificar explícitamente un literal de tipo **double** añadiendo una *D* o *d*. Hacerlo así es, evidentemente, redundante. El tipo **double** por omisión consume 64 bits para el almacenamiento, mientras que el tipo **float** es menos exacto y requiere únicamente 32 bits.

Literales booleanos

Los literales booleanos son sencillos. Existen sólo dos valores lógicos que puede tener un valor del tipo **boolean**, que son los valores **true** y **false**. Estos valores no se convierten en ninguna representación numérica. El literal **true** en Java no es igual a 1, ni el **false** igual a 0. En Java, estos dos literales solamente se pueden asignar a variables declaradas como **boolean**, o utilizadas en expresiones con operadores booleanos.

Literales de tipo carácter

Los caracteres de Java son índices dentro del conjunto de caracteres Unicode. Son valores de 16 bits que pueden ser convertidos en enteros y manipulados con operadores enteros como los operadores de suma y resta. Un literal de carácter se representa dentro de una pareja de comillas simples. Todos los caracteres ASCII visibles se pueden introducir directamente dentro de las comillas, como por ejemplo,'*a*','*z*', y'@'. Para los caracteres que resulta imposible introducir directamente, existen varias secuencias de escape que permiten introducir al carácter deseado como'\'' para el propio carácter de comilla simple, y '**\n**' para el carácter de línea nueva. También existe un mecanismo para introducir directamente el valor de un carácter en base octal o hexadecimal. Para la notación octal se utiliza la diagonal invertida seguida por el número de tres dígitos. Por ejemplo,'*141*' es la letra'*a*'. Para la notación hexadecimal, se escribe la diagonal invertida seguida de una u (**\u**), y exactamente cuatro dígitos hexadecimales. Por ejemplo,'*u0061*' es el carácter ISO-Latin-1'*a*', ya que el bit superior es cero.'*ua432*' es un carácter japonés Katakana. La Tabla 3-1 muestra las secuencias de caracteres de escape.

Literales de tipo cadena

Los literales de tipo cadena en Java se especifican como en la mayoría de los lenguajes, encerrando la secuencia de caracteres en una pareja de comillas dobles. Por ejemplo,

```
"Hola Mundo"
"dos \nlíneas"
"\"Esto está entre comillas\""
```

TABLA 3-1
Secuencias de escape

Secuencia de escape	Descripción
\ddd	Carácter escrito en base octal (ddd)
\uxxxx	Carácter escrito utilizando su valor Unicode en hexadecimal (xxxx)
\’	Comilla simple
\”	Comilla doble
\\	Diagonal
\r	Retorno de carro
\n	Nueva línea o salto de línea
\f	Comienzo de página
\t	Tabulador
\b	Retroceso

Las secuencias de escape y la notación octal/hexadecimal definidas para caracteres literales funcionan del mismo modo en las cadenas de literales. Una cuestión importante, respecto a las cadenas en Java, es que deben comenzar y terminar en la misma línea. No existe, como en otros lenguajes, una secuencia de escape para continuación de la línea.

NOTA *Como sabrá, en la mayoría de los lenguajes, incluyendo C/C++, las cadenas se implementan como arreglos de caracteres. Sin embargo, éste no es el caso en Java. Las cadenas son realmente un tipo de objetos. Como se verá posteriormente, ya que Java implementa las cadenas como objetos, incluye un extensivo conjunto de facilidades para manejo de cadenas que son, a la vez, potentes y fáciles de manejar.*

Variables

La variable es la unidad básica de almacenamiento en un programa Java. Una variable se define mediante la combinación de un identificador, un tipo y un inicializador opcional. Además, todas las variables tienen un ámbito que define su visibilidad y tiempo de vida. A continuación se examinan estos elementos.

Declaración de una variable

En Java, se deben declarar todas las variables antes de utilizarlas. La forma básica de declaración de una variable es la siguiente:

tipo identificador [= *valor*][, *identificador* [= *valor*] ...];

El *tipo* es uno de los tipos de Java, o el nombre de una clase o interfaz. (Los tipos de clases e interfaces se analizan más adelante, en la Parte 1 de este libro). El *identificador* es el nombre de la variable. Se puede inicializar la variable mediante un signo igual seguido de un valor. Tenga en cuenta que la expresión de inicialización debe dar como resultado un valor del mismo tipo (o de un tipo compatible) que el especificado para la variable. Para declarar más de una variable del tipo especificado, se utiliza una lista con los elementos separados por comas.

A continuación se presentan ejemplos de declaraciones de variables de distintos tipos. Observe cómo algunas de estas declaraciones incluyen una inicialización.

```
int a, b, c;            // declara tres enteros, a, b, y c.
int d = 3, e, f = 5;    // declara tres enteros más, inicializando d y f.

byte z = 22;            // inicializa z.
double pi = 3.14159;    // declara una aproximación de pi.
char x = 'x';           // la variable x tiene el valor 'x'.
```

Los identificadores elegidos no tienen nada intrínseco en sus nombres que indique su tipo. Java permite que cualquier nombre correcto sea utilizado para declarar una variable de cualquier tipo.

Inicialización dinámica

Aunque los ejemplos anteriores han utilizado únicamente constantes como inicializadores, Java permite la inicialización dinámica de variables mediante cualquier expresión válida en el instante en que se declara la variable.

A continuación se presenta un programa corto que calcula la longitud de la hipotenusa de un triángulo rectángulo a partir de la longitud de los dos catetos.

```
// Ejemplo de inicialización dinámica.
class DynInit {
    public static void main(String args[]) {
      double a = 3.0, b = 4.0;

      // Se inicializa c dinámicamente
      double c = Math.sqrt(a * a + b * b);

      System.out.println"La hipotenusa es " + c);
    }
}
```

En este ejemplo se declaran tres variables locales, **a**, **b** y **c**. Las dos primeras, **a** y **b**, se han inicializado mediante constantes; sin embargo, **c** se inicializa dinámicamente como la longitud de la hipotenusa; calculada mediante el teorema de Pitágoras. El programa utiliza otro de los métodos definidos en Java, **sqrt( )**, que es un miembro de la clase **Math**, para calcular la raíz cuadrada de su argumento. El punto clave aquí es que la expresión de inicialización puede usar cualquier elemento válido en el instante de la inicialización, incluyendo la llamada a métodos, otras variables, o literales.

Ámbito y tiempo de vida de las variables

Hasta el momento, todas las variables que se han utilizado se han declarado al comienzo del método **main**( ). Sin embargo, Java permite la declaración de variables dentro de un bloque. Tal y como se vió en el Capítulo 2, un bloque comenzaba con una llave de apertura y terminaba con una llave de cierre. Un bloque define un ámbito. Cada vez que se inicia un nuevo bloque, se está creando un nuevo ámbito. Un ámbito determina qué objetos son visibles para otras partes del programa. También determina el tiempo de vida de esos objetos.

La mayoría de lenguajes definen dos categorías generales de ámbitos: global y local. Sin embargo, estos ámbitos tradicionales no se ajustan estrictamente al modelo orientado a objetos de Java. En Java, los dos grandes ámbitos son el definido por las clases, y el definido por los

métodos. Esta distinción es, de alguna manera, artificial. Sin embargo, esta distinción tiene sentido, ya que el ámbito de la clase tiene ciertas propiedades y atributos que no se pueden aplicar al ámbito definido por un método. Teniendo en cuenta estas diferencias, se deja para el Capítulo 6, en el que se describen las clases, la discusión sobre el ámbito de las clases y las variables declaradas dentro de una clase. De momento sólo examinaremos los ámbitos definidos por un método o en un método.

El ámbito definido por un método comienza con la llave que inicia el cuerpo del método. Si el método tiene parámetros, éstos también están incluidos en el ámbito del método. Aunque los parámetros se analizan con más profundidad en el Capítulo 6, en este momento se puede decir que son equivalentes a cualquier otra variable del método.

Como regla general, se puede decir que las variables declaradas dentro de un ámbito no son visibles, es decir, accesibles, al código definido fuera de ese ámbito. Por tanto, cuando se declara una variable dentro de un ámbito, se está localizando y protegiendo esa variable contra un acceso no autorizado y/o modificación. Las reglas de ámbito proporcionan la base de la encapsulación.

Los ámbitos pueden estar anidados. Por ejemplo, cada vez que se crea un bloque de código, se está creando un nuevo ámbito anidado. Cuando esto sucede, los ámbitos exteriores encierran al ámbito interior. Esto significa que los objetos declarados en el ámbito exterior son visibles para el código dentro del ámbito interior. Sin embargo, no ocurre igual en el sentido opuesto, los objetos declarados en el ámbito interior no son visibles fuera del mismo.

Para entender el efecto de los ámbitos anidados, consideremos el siguiente programa:

```
// Ejemplo de ámbito de un bloque.
class Ambito {
  public static void main(String args[]) {
    int x; // conocida para todo el código que está dentro de main

    x = 10;
    if(x == 10) ( // comienzo de un nuevo ámbito
      int y = 20; // conocida solamente dentro de este bloque

      // aquí, se conocen tanto x como y.
      System.out.println("x e y: " + x + " " + y);
      x = y * 2;
    }
    // y = 100; // Error! Aquí no se conoce y

    // aquí todavía se conoce x.
    System.out.println("x es " + x);
  }
}
```

Como lo indican los comentarios, la variable **x** se declara al comienzo del ámbito del método **main( )** y es accesible para todo el código contenido dentro de **main( )**. Dentro del bloque **if** se declara **y**. Como un bloque define un ámbito, **y** sólo es visible para el código que está dentro de su bloque. Por ello, fuera de su bloque, la línea **y = 100**; tuvo que ser precedida por el símbolo de comentario. Si se elimina este comentario, se producirá un error de compilación, ya que la variable y no es visible fuera de su bloque. Sin embargo, dentro del bloque **if** se puede utilizar la variable **x**, ya que dentro de un bloque (un ámbito anidado) se tiene acceso a las variables declaradas en un ámbito exterior.

Dentro de un bloque, las variables se pueden declarar en cualquier punto, pero sólo son válidas después de ser declaradas. Por tanto, si se define una variable al comienzo de un método, está disponible para todo el código contenido en el método. Por el contrario, si se declara una variable al final de un bloque, ningún código tendrá acceso a la misma. El siguiente fragmento de código no es válido, ya que no se puede usar la variable **count** antes de su declaración:

```
// Este fragmento no es correcto!
count = 100; // No se puede utilizar count antes de declararla
int count;
```

Otro punto importante que se ha de tener en cuenta es el siguiente: las variables se crean cuando la ejecución del programa alcanza su ámbito, y son destruidas cuando se abandona su ámbito. Esto significa que una variable no mantiene su valor una vez que se ha salido de su ámbito. Por tanto, las variables declaradas dentro de un método no mantienen sus valores entre llamadas a ese método. Del mismo modo, una variable declarada dentro de un bloque pierde su valor cuando se abandona el bloque. Es decir, el tiempo de vida de una variable está limitado por su ámbito.

Si la declaración de una variable incluye un inicializador, entonces esa variable se reinicializa cada vez que se entra en el bloque en que ha sido declarada. Consideremos el siguiente programa:

```
// Ejemplo del tiempo de vida de una variable.
c1ass Duracion {
  public static void main(String args[]) {
    int x;

    for(x = 0; x < 3; x++) {
      int y = -1; // y se inicializa cada vez que se entra en el bloque
      System.out.println("y es: " + y); // siempre se imprime -1
      y = 100;
      System.out.println("y es ahora: " + y);
    }
  }
}
```

La salida generada por este programa es la siguiente:

```
y es: -1
y es ahora: 100
y es: -1
y es ahora: 100
y es: -1
y es ahora: 100
```

Como puede verse, cada vez que se entra en el ciclo **for** interior, la variable **y** se reinicializa a –1. Aunque a continuación se le asigne el valor 100, este valor se pierde.

Por último, aunque los bloques pueden estar anidados, no se puede declarar una variable con el mismo nombre que otra que está en un ámbito exterior. El ejemplo que se presenta a continuación muestra una declaración no válida de variables:

```
// Este programa no se compilará
class AmbitoErr {
```

```
    public static void main(String args[]) {
      int bar = 1;
      { // se crea un nuevo ámbito
      int bar = 2; // Error de compilación, ¡la variable bar ya está definida!
    }
  }
}
```

Conversión de tipos

Si tiene cierta experiencia en programación, ya sabrá que es bastante común asignar un valor de un tipo a una variable de otro tipo. Si los dos tipos son compatibles, Java realiza la conversión automáticamente. Por ejemplo, siempre es posible asignar un valor del tipo **int** a una variable del tipo **long**. Sin embargo, no todos los tipos son compatibles, y, por lo tanto, no cualquier conversión está permitida implícitamente. Por ejemplo, la conversión de **double** a **byte** no está definida. Afortunadamente, se puede obtener una conversión entre tipos incompatibles. Para ello, se debe usar un **cast**, que realiza una conversión explícita entre tipos. Veamos ambos tipos de conversión.

Conversiones automáticas de Java

Cuando datos de un tipo se asignan a una variable de otro tipo, tiene lugar una conversión automática de tipo si se cumplen las siguientes condiciones:

- Los dos tipos son compatibles.
- El tipo destino es más grande que el tipo fuente.

Cuando se cumplen estas dos condiciones, se produce una *conversión de ensanchamiento* o promoción. Por ejemplo, el tipo **int** siempre es lo suficientemente amplio para almacenar todos los valores válidos del tipo **byte**, de manera que se realiza una conversión automática.

En este tipo de conversiones, los tipos numéricos, incluyendo los tipos enteros y de punto flotante, son compatibles entre sí. Sin embargo, los tipos numéricos no son compatibles con los tipos **char** o **boolean**. Además, **char** y **boolean** no son compatibles entre sí.

Como se mencionó anteriormente, Java también realiza una conversión automática de tipos cuando se almacena una constante entera en variables del tipo **byte, short, long** o **char**.

Conversión de tipos incompatibles

Aunque la conversión automática de tipos es útil, no es capaz de satisfacer todas las necesidades. Por ejemplo, ¿qué ocurre si se quiere asignar un valor del tipo **int** a una variable del tipo **byte**? Esta conversión no se realiza automáticamente porque un valor del tipo **byte** es más pequeño que un valor del tipo **int**. Esta clase de conversión se denomina en ocasiones estrechamiento, ya que explícitamente se *estrecha el valor* para que se ajuste al tipo de destino.

Para realizar una conversión entre dos tipos incompatibles, se debe usar un cast. Un **cast** es simplemente una conversión de tipos explícita, y tiene la siguiente forma genérica:

(*tipo*) *valor*

Donde tipo especifica el tipo al que se desea convertir el valor especificado. Por ejemplo, el siguiente fragmento convierte un **int** en un **byte**. Si el valor del entero es mayor que el rango de un **byte**, se reducirá al módulo (residuo de la división entera) del rango del tipo **byte**.

```
int a;
byte b;
// ...
b = (byte) a;
```

Una conversión diferente es la que tiene lugar cuando se asigna un valor de punto flotante a un tipo entero. En este caso, se *trunca* la parte fraccionaria. Como ya se sabe, los enteros no tienen componente fraccional. Por tanto, cuando se asigna un valor en punto flotante a un entero, se pierde la componente fraccional. Por ejemplo, si se asigna a un entero el valor 1.23, el valor resultante será simplemente 1, truncándose la parte fraccionaria, 0.23. Naturalmente, si el tamaño de la componente numérica es demasiado grande para ajustarse al tipo entero de destino, entonces ese valor se reducirá al módulo del rango del tipo de destino.

El siguiente programa ejemplifica algunas conversiones de tipo explícitas:

```
// Ejemplo de conversiones de tipo explícitas (cast)
class Conversion {
  public static void main(String args[]) {
    byte b;
    int i = 257;
    double d = 323.142;

    System.out.println("\nConversión de int a byte.");
    b = (byte) i;
    System.out.println{"i y b " + i + " " + b);

    System.out.println("\nConversión de double a int.");
    i = (int) d;
    System.out.println{"d y i " + d + " " + i);

    System.out.println{"\nConversión de double a byte.");
    b = (byte) d;
    System.out.println("d y b " + d + " " + b);
  }
}
```

La salida que produce este programa es:

```
Conversión de int a byte.
i y b 257 1

Conversión de double a int.
d y i 323.142 323

Conversión de double a byte.
d y b 323 .142 67
```

Veamos cada una de estas conversiones. Cuando se convierte el valor 257 a una variable **byte**, el resultado es el residuo de la división entera de 257 entre 256 (el rango del tipo **byte**), que en este caso es 1. Cuando se convierte **d** en **int**, se pierde su componente fraccional. Cuando se convierte **d** a **byte**, se pierde su componente fraccionaria y se reduce su valor al módulo de 256, que en este caso es 67.

Promoción automática de tipos en las expresiones

Las conversiones de tipo, además de ocurrir en la asignación de valores, pueden tener lugar en las expresiones. Para ver cómo sucede esto, consideremos el siguiente caso. En una expresión, puede ocurrir que la precisión requerida por un valor intermedio exceda el rango de cualquiera de los operandos. Por ejemplo, en la siguiente expresión:

```
byte a = 40;
byte b = 50;
byte e = 100;
int d = a * b / c;
```

El resultado del término intermedio **a * b** excede fácilmente el rango de sus operandos, que son del tipo **byte**. Para resolver este tipo de problema, Java convierte automáticamente cada operando del tipo **byte** o **short** al tipo **int**, al evaluar una expresión. Esto significa que la subexpresión a *** b** se calcula utilizando tipos enteros, no bytes. Por tanto, el resultado de la operación intermedia, **50 * 40**, es válido aunque se hayan especificado **a** y **b** como del tipo **byte**.

Aunque las promociones automáticas son muy útiles, pueden dar lugar a errores confusos en tiempo de compilación. Por ejemplo, este código, aparentemente correcto, ocasiona un problema:

```
byte b = 50;
b = b * 2; / / Error, ¡no se puede asignar un int a un byte!
```

Este código intenta almacenar 50 * 2, un valor del tipo **byte** perfectamente válido, en una variable **byte**. Sin embargo, cuando se evaluó la expresión, los operandos fueron promocionados automáticamente al tipo **int**. Por tanto, el tipo de la expresión es ahora del tipo **int**, y no se puede asignar al tipo **byte** sin utilizar la conversión explícita. Esto ocurre incluso si, como en este caso, el valor que se intenta asignar está en el rango del tipo objetivo.

En casos como el siguiente, en que se prevén las consecuencias del desbordamiento, se debería usar la conversión explícita,

```
byte b = 50;
b = (byte) (b * 2);
```

que conduce al valor correcto de 100.

Reglas de la promoción de tipos

Java define varias reglas *para la promoción* de tipos que se aplican a las expresiones. Estas reglas son las siguientes. En primer lugar, los valores **byte**, **short** y **char** son promocionados al tipo **int**, como se acaba de describir. Además, si un operando es del tipo **long**, la expresión completa es promocionada al tipo **long**. Si un operando es del tipo **float**, la expresión completa es promocionada al tipo **float**. Si cualquiera de los operandos es **double**, el resultado será **double**.

El siguiente programa muestra cómo se promociona cada valor en la expresión para coincidir con el segundo argumento de cada operador binario:

```
class Promocion {
  public static void main(String args[]) {
    byte b = 42;
    char c ='a';
    short s = 1024;
```

```
    int i = 50000;
    float f = 5.67f;
    double d = .1234;
    double resultado = (f * b) + (i / e) - (d * s);
    System.out.println((f * b) + " + " + (i / e) + " - " + (d * s));
    System.out.println("resultado = " + resultado);
  }
}
```

Examinemos con más detalle todas las promociones de tipos que tienen lugar en esta línea del programa:

```
double result = (f * b) + (i / c) - (d * s);
```

En la subexpresión, **f * b, b** es promocionado a **float** y el resultado de la subexpresión es del tipo **float**. A continuación, en la subexpresión **i / c, c** es promocionado a **int**, y el resultado es del tipo **int**. Luego, en **d * s**, el valor de **s** se promociona a **double**, y el tipo de la expresión es **double**. Finalmente, considerando estos tres valores intermedios, **float**, **int** y **double**, el resultado de **float** más un **int** es del tipo **float**. A continuación, el resultado de un **float** menos el último **double** es promocionado a **double**, que es el tipo final del resultado de la expresión.

Arreglos

Un *arreglo* es un grupo de variables del mismo tipo al que se hace referencia por medio de un nombre común. Se pueden crear arreglos de cualquier tipo, y pueden tener una dimensión igual a uno o mayor. Para acceder a un elemento concreto de un arreglo se utiliza su índice. Los arreglos ofrecen un medio conveniente para agrupar información relacionada.

NOTA *Si está familiarizado con C/C++, debe tener cuidado. Los arreglos, en Java, funcionan de forma diferente a como funcionan los arreglos en esos dos lenguajes.*

Arreglos unidimensionales

Un *arreglo unidimensional* es, esencialmente, una lista de variables del mismo tipo. Para crear un arreglo, primero se debe crear una variable arreglo del tipo deseado. La forma general de declarar un arreglo unidimensional es:

tipo nombre [ ];

En donde, *tipo* declara el tipo base del arreglo, el cual determina el tipo de cada elemento que conforma el arreglo. Por lo tanto, el tipo base determina qué tipo de datos almacenará el arreglo. Por ejemplo, la siguiente línea declara un arreglo llamado **dias_del_ mes** con el tipo"arreglo de int":

```
int dias_del_mes[];
```

Aunque esta declaración establece que **dias_del _mes** es una variable de tipo arreglo, todavía no existe realmente ningún arreglo. De hecho, el valor de **dias_del_mes** es **null**, null

representa un arreglo que no tiene ningún valor. Para que **dias_del_mes** sea un verdadero arreglo de enteros se debe reservar espacio utilizando el operador **new** y asignar este espacio a **dias_del_mes**. **new** es un operador especial que reserva espacio de memoria.

Este operador se verá con más detalle en un capítulo posterior, pero es preciso utilizarlo en este momento para reservar espacio para los arreglos. La forma general del operador **new** cuando se aplica a arreglos unidimensionales es la siguiente:

nombre = new *tipo*[*tamaño*];

donde *tipo* especifica el tipo de datos almacenados en el arreglo, *tamaño* especifica el número de elementos, el *arreglo y nombre* es la variable a la que se asigna el nuevo arreglo; es decir, al usar **new** para reservar espacio para un arreglo, se debe especificar el tipo y número de elementos que se van a almacenar. Al reservar espacio para los elementos del arreglo mediante **new**, todos los elementos se inicializan a cero automáticamente. El siguiente ejemplo reserva espacio para un arreglo de 12 elementos enteros y los asigna a **dias_del_mes**.

```
dias_del_mes = new int[12];
```

Cuando se ejecute esta sentencia, **dias_del_mes** hará referencia a un arreglo de 12 elementos enteros. Además, todos los elementos del arreglo se inicializan a cero.

Resumiendo, la obtención de un arreglo es un proceso que consta de dos partes. En primer lugar, se debe declarar una variable del tipo de arreglo deseado. En segundo lugar, se debe reservar espacio de memoria para almacenar el arreglo mediante el operador **new**, y asignarlo a la variable. En Java, la memoria necesaria para los arreglos se reserva dinámicamente. Si no le resulta familiar el concepto de reserva dinámica, no se preocupe, se describirá con detalle más adelante en este libro.

Una vez reservada la memoria para un arreglo, se puede acceder a un elemento concreto del arreglo especificando su índice dentro de corchetes. Todos los índices de un arreglo comienzan en cero. Por ejemplo, la siguiente sentencia asigna el valor 28 al segundo elemento de **dias_del_mes**.

```
dias_del_mes[1] = 28;
```

La siguiente línea muestra el valor correspondiente al índice 3.

```
System.out.println(dias_del_mes[3]);
```

El siguiente programa resume las ideas anteriores, creando un arreglo con el número de días de cada mes.

```
// Ejemplo de un arreglo unidimensional.
class Arreglo {
  public static void main(String args[]) {
    int dias_del_mes [];
    dias_del_mes =new int[12];
    dias_del_mes [0] = 31;
    dias_del_mes [1] = 28;
    dias_del_mes [2] = 31;
    dias_del_mes [3] = 30;
    dias_del_mes [4] = 31;
    dias_del_mes [5] = 30;
```

```
    dias_del_mes [6] = 31;
    dias_del_mes [7] = 31;
    dias_del_mes [8] = 30;
    dias_del_mes [9] = 31;
    dias_del_mes [10] = 30;
    dias_del_mes [11] = 31;
    System.out.println("Abril tiene" + dias_del_mes [3] + " días.");
  }
}
```

Cuando se ejecuta este programa, se imprime el número de días que tiene el mes de Abril. Como se ha indicado, los índices de arreglos en Java, comienzan en cero, es decir, el número de días del mes de Abril es **dias_del_mes** [**3**] o 30.

Es posible combinar la declaración de una variable de tipo arreglo con la reserva de memoria para el propio arreglo, tal y como se muestra a continuación:

```
int dias_del_mes [] = new int [12];
```

Ésta es la declaración que normalmente se hace en los programas profesionales escritos en Java.

Los arreglos también pueden ser inicializados cuando se declaran. El proceso es el mismo que cuando se inicializan tipos sencillos. Un *inicializador de arreglo* es una lista de expresiones entre llaves separadas por comas. Las comas separan los valores de los elementos del arreglo. Se creará un arreglo lo suficientemente grande para que pueda contener los elementos que se especifiquen en el inicializador del arreglo. No es necesario utilizar **new**. Por ejemplo, el siguiente código crea un arreglo de enteros para almacenar el número de días de cada mes:

```
// Versión mejorada del programa anterior.
class AutoArreglo (
  public static void main(String args[]) {

    int dias_del_mes [] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

    System.out.println("Abril tiene" + dias_del_mes [3] +" días.");
  }
}
```

Cuando se ejecuta este programa, se obtiene la misma salida que generó la versión anterior.

Java comprueba estrictamente que no se intente almacenar o referenciar accidentalmente valores que estén fuera del rango del arreglo. El intérprete de Java comprueba que todos los índices del arreglo están dentro del rango correcto. Por ejemplo, el intérprete de Java comprobará el valor de cada índice en **dias_del_mes** para asegurar que se encuentran comprendidos entre 0 y 11. Si se intenta acceder a elementos que estén fuera del rango del arreglo (con índices negativos o índices mayores que el rango del arreglo), se producirá un error en tiempo de ejecución.

En el siguiente ejemplo se utiliza un arreglo unidimensional para calcular el valor promedio de un conjunto de números.

```
// Promedia los valores de un arreglo
class Promedio (
  public static void main(String args[]) {
    double nums[] = {10.1, 11.2, 12.3, 13.4, 14.5};
    double resultado = 0;
    int i;
```

```
    for(i=0; i<5; i++)
      resultado = resultado + nums[i];

    System.out.println("La media es " + resultado / 5);
  }
}
```

Arreglos multidimensionales

En Java, los *arreglos multidimensionales* son realmente arreglos de arreglos. Tal y como se podría esperar, se parecen a los arreglos multidimensionales y actúan como estos. Sin embargo, como se verá, existen un par de sutiles diferencias. Para declarar una variable de arreglo multidimensional, hay que especificar cada índice adicional utilizando otra pareja de corchetes. Por ejemplo, la siguiente línea declara una variable de arreglo bidimensional denominada **dosD**.

```
int dosD[] [] = new int[4] [5];
```

Esta sentencia reserva espacio para un arreglo de dimensión 4 por 5, y la asigna a **dosD.** Internamente esta matriz se implementa como un *arreglo* de *arreglos* del tipo **int**. Conceptualmente, este arreglo se parece al mostrado en la Figura 3.1.

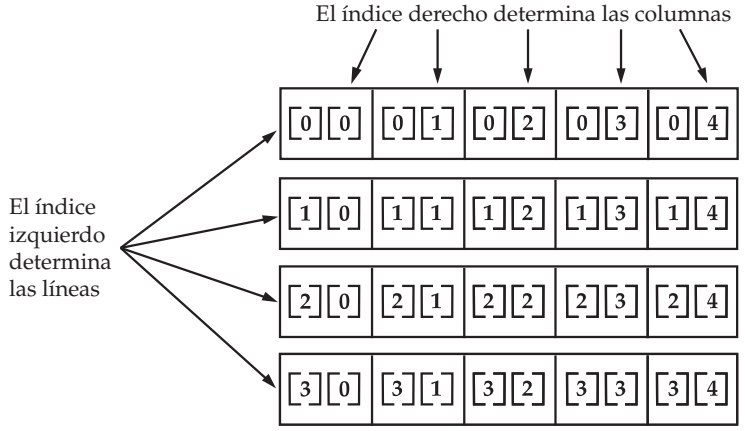
El siguiente programa asigna un valor numérico a cada elemento del arreglo de izquierda a derecha y de arriba abajo, y a continuación despliega esos valores.

```
// Ejemplo de un arreglo bidimensional.
class ArregloDosD {
  public static void main(String args[]) {
    int dosD[] []= new int[4] [5];
    int i, j, k = 0;

    for(i=0; i<4; i++)
      for{j=0; j<5; j++) {
        dosD[i] [j] =k;
        k++;

      }

    for(i=0; i<4; i++) {
      for(j=0; j<5; j++)
        System.out.print(dosD[i] [j] + " ");
      System.out.println();
    }
  }
}
```

Este programa produce la siguiente salida:

```
0 1 2 3 4
5 6 7 8 9
10 11 12 13 14
15 16 17 18 19
```

Cuando se reserva memoria para un arreglo multidimensional, sólo es necesario especificar la memoria para la primera dimensión, es decir, la que está más a la izquierda.



Dado: int dosD[][] = int nuevo [4][5];

FIGURA 3.1 Una vista conceptual de un arreglo bidimensional de 4 x 5.

Después se puede reservar la memoria para las restantes dimensiones. El siguiente código reserva memoria para la primera dimensión de **dosD** cuando se declara la variable. El espacio para la segunda dimensión se reserva manualmente.

```
int dosD[] [] =new int[4] [];
dosD [0] = new int[5];
dosD [1] = new int[5];
dosD [2] = new int[5];
dosD [3] = new int[5];
```

En este caso no resulta ventajoso reservar espacio para la segunda dimensión individualmente, sin embargo, en otros casos puede que sí lo sea. Por ejemplo, cuando se reserva memoria para varias dimensiones de forma separada, no es necesario reservar el mismo número de elementos para cada dimensión. Como se ha indicado anteriormente, al ser los arreglos multidimensionales realmente arreglos de arreglos, la longitud de cada uno se puede establecer de forma independiente. El siguiente programa crea un arreglo bidimensional en la que los tamaños de la segunda dimensión no son iguales.

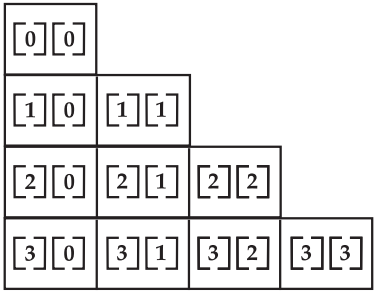
```
// Reserva de diferentes tamaños para la segunda dimensión.
class OtroDosD {
  public static void main(String args[]) {
    int dosD[] [] =new int[4][];.
    dosD [0] = new int[l];
    dosD [1] = new int[2];
    dosD [2] = new int[3];
    dosD [3] = new int[4];

    int i, j, k = 0;

    for(i=0; i<4; i++)
```

```
      for(j=0; j<i+l; j++) {
        dosD [i] [j] =k;
        k++;
      }
    for(i=0; i<4; i++) {
      for(j=0; j<i+l; j++)
        System.out.print(dosD [i][j] + " ");
      System.out.println();
    }
  }
}
```

Este programa genera la siguiente salida:

```
0
1 2
3 4 5
6 7 8 9
```

El arreglo creado por este programa tiene una forma parecida a ésta:



En muchas aplicaciones no se recomienda el uso de arreglos multidimensionales irregulares, porque no coincide con lo que mucha gente supone al encontrar un arreglo multidimensional. Sin embargo, estos arreglos se pueden utilizar efectivamente en muchas situaciones. Por ejemplo, si se necesita un arreglo bidimensional de gran tamaño pero que esté dispersamente poblado, es decir, en el que no se van a utilizar todos sus elementos, un arreglo irregular puede ser la solución ideal.

Se puede inicializar un arreglo multidimensional. Para ello, se encierra entre llaves el inicializador de cada dimensión. El siguiente programa crea una matriz en la que cada elemento contiene el producto de los índices de la fila y columna. Observe también que se pueden utilizar expresiones y literales en los inicializadores de arreglos.

```
// Inicialización de un arreglo bidimensional.
class Matriz {
  public static void main(String args[]) {
    double m[] [] = {
      { 0*0, 1*0, 2*0, 3*0 },
      { 0*1, 1*1, 2*1, 3*1 },
      { 0*2, 1*2, 2*2, 3*2 },
      { 0*3, 1*3, 2*3, 3*3 }
```

```
    };
    int i, j;

    for(i=0; i<4; i++) {
      for(j=0; j<4; j++)
        System.out.print(m[i][j] + " ");
      System.out.println();
    }
  }
}
```

Cuando se ejecuta este programa se obtiene la siguiente salida:

```
0.0   0.0   0.0   0.0
0.0   1.0   2.0   3.0
0.0   2.0   4.0   6.0
0.0   3.0   6.0   9.0
```

Como se puede ver, cada fila del arreglo se inicializa según se especifica en las listas de incialización.

Veamos un ejemplo más, en el que se utiliza un arreglo multidimensional. El siguiente programa crea un arreglo tridimensional de 3 por 4 por 5. A continuación almacena en cada elemento el producto de los índices correspondiente. Finalmente despliega estos productos.

```
// Ejemplo de un arreglo tridimensional.
class MatrizTresD (
  public static void main(String args[]) {
    int tresD[] [] [] =new int[3] [4] [5];
    int i. j, k;

    for(i=0; i<3; i++)
      for(j=0; j<4; j++)
        for(k=0; k<5; k++)
          tresD[i] [j] [k] = i * j * k;

    for(i=0; i<3; i++) {
      for(j=0; j<4; j++} {
        for(k=0; k<5; k++)
          System.out.print(tresD[i][j][k] + " ");
        System.out.println();
      }
      System.out.println();
    }
  }
}
```

Este programa produce la siguiente salida:

```
0  0  0  0  0
0  0  0  0  0
0  0  0  0  0
0  0  0  0  0
0  0  0  0  0
0  1  2  3  4
0  2  4  6  8
0  3  6  9  12
```

```
0  0  0  0  0
0  2  4  6  8
0  4  8  12  16
0  6  12  18  24
```

Sintaxis alternativa para la declaración de arreglos

Existe una segunda forma de declarar un arreglo:

tipo[ ] *nombre;*

Aquí los corchetes siguen al especificador de tipo, y no al nombre del arreglo. Por ejemplo, las dos declaraciones siguientes son equivalentes:

```
int a1[] = new int[3];
int[] a2 = new int[3];
```

Las siguientes declaraciones también son equivalentes:

```
char dosd1[] [] = new char[3] [4];
char[] [] dosd2 = new char[3] [4];
```

Esta forma alternativa de declaración resulta conveniente cuando se declaran varios arreglos al mismo tiempo, por ejemplo:

```
int []nums, nums2, nums3; // crea tres arreglos
```

Crea tres variables de tipo arreglo de **int**. La declaración anterior equivale a escribir

```
int nums[], nums2[], nums3[]; // crea tres arreglos
```

La forma alternativa de declaración también es útil cuando se especifica un arreglo como tipo de retorno para un método. Ambas formas se utilizan en este libro.

Unas breves notas sobre las cadenas

Como habrá podido observar en el análisis anterior sobre los tipos de datos y arreglos, no se ha hecho ninguna mención sobre las cadenas o el tipo de dato cadena. Esto no se debe a que Java no soporte este tipo, sino a que el tipo de cadena de Java, denominado **String**, no es un tipo simple. No se trata simplemente de un arreglo de caracteres. En lugar de esto, **String** define un objeto, y para hacer una descripción completa es preciso comprender algunas de las características de los objetos, que se verán más adelante, cuando se describan los objetos. Sin embargo, hacemos en este momento una breve introducción para poder utilizar en los programas de ejemplo algunas cadenas sencillas.

El tipo **String** se utiliza para declarar variables de tipo cadena. También se pueden declarar arreglos de cadenas. A una variable de tipo **String** se le puede asignar una constante del tipo cadena delimitada por comillas.

También se le puede asignar otra variable del tipo **String**. Se puede utilizar un objeto del tipo **String** como un argumento de **println( )**. Consideremos el siguiente fragmento de código:

```
String str = "esto es una prueba";
System.out.println(str) ;
```

Aquí **str** es un objeto del tipo **String** al que se asigna la cadena"esto es una prueba". Esta cadena se imprime mediante la sentencia **println( ).**

Como se verá más adelante, los objetos **String** tienen características y atributos especiales que les hacen potentes y fáciles de utilizar. Sin embargo, en los próximos capítulos sólo se utilizaran en su forma más sencilla.

Una nota para los programadores de C/C++ sobre los apuntadores

Si usted es un programador con experiencia en C/C++, entonces sabe que estos dos lenguajes permiten el uso de apuntadores. Sin embargo, en este capítulo no se ha hecho ninguna mención a los apuntadores. La razón es sencilla: Java no permite la utilización de apuntadores o, dicho de forma más precisa, Java no permite apuntadores a los que el programador pueda acceder o modificar. Java no puede permitir apuntadores, ya que si lo hiciera, los programas de Java podrían romper la barrera existente entre el entorno de ejecución de Java y la computadora. Recuerde que a un apuntador se le puede dar cualquier dirección de memoria, incluso las que estuviesen fuera del intérprete Java. Como C/C++ hace un uso extensivo de los apuntadores, se podría pensar que su pérdida es una desventaja. Sin embargo, esto no es así. Java está diseñado de forma que, en tanto se esté dentro de los límites del entorno de ejecución, no se necesitará el uso de apuntadores, ni se obtendría ningún beneficio al utilizarlos.

CAPÍTULO 4

Operadores

Java proporciona un amplio conjunto de operadores, que se pueden dividir en los cuatro grupos siguientes: aritméticos, a nivel de bit, relacionales y lógicos. Java también define algunos operadores adicionales para la gestión de situaciones especiales. Este capítulo describe todos los operadores de Java, excepto el operador de comparación de tipo **instanceof**, que se estudia en el Capítulo 13.

Operadores aritméticos

Los operadores aritméticos se utilizan en expresiones matemáticas de la misma forma que son utilizados en álgebra. En la siguiente tabla se da un listado de los operadores aritméticos:

Operador	Resultado
+	Suma
-	Resta (también es el menos unario)
*	Multiplicación
/	División
%	Módulo
++	Incremento
+=	Suma y asignación
–=	Resta y asignación
*=	Multiplicación y asignación
/=	División y asignación
%=	Módulo y asignación
– –	Decremento

Los operandos de los operadores aritméticos deben ser del tipo numérico. No se pueden utilizar sobre operandos del tipo **boolean**, pero sí sobre operadores del tipo **char**, ya que el tipo **char** de Java es, esencialmente, un subconjunto de **int**.

Operadores aritméticos básicos

Las operaciones aritméticas básicas —suma, resta, multiplicación y división— se comportan con todos los tipos numéricos como se espera. El operador menos también tiene una forma unaria que sirve para negar su operando único. Recuerde que cuando se aplica el operador de división a un tipo entero no existirá componente decimal en el resultado.

El siguiente programa ejemplifica el funcionamiento de los operadores aritméticos. También muestra la diferencia entre la división de números de punto flotante y la división de enteros.

```
// Ejemplo del funcionamiento de los operadores aritméticos básicos.
c1ass BasicMat {
  pub1ic static void main(String args[]) {
    // aritmética utilizando enteros
    System.out.print1n("Aritmética de Enteros");
    int a = 1 + 1;
    int b = a * 3;
    int c = b / 4;
    int d = c - a;
    int e = - d;
    System.out.print1n("a = " +a);
    System.out.print1n("b = " +b);
    System.out.print1n("c = " +c);
    System.out.print1n("d = " +d);
    System.out.print1n("e = " +e);

    // Aritmética utilizando el tipo doble
    System.out.println("\nAritmética de Punto F1otante");
    doub1e da = 1 + 1;
    doub1e db = da * 3;
    doub1e dc = db / 4;
    doub1e dd = dc - a;
    doub1e de = -dd;
    System.out.print1n("da = " + da);
    System.out.print1n("db = " + db);
    System.out.print1n("dc = " + dc);
    System.out.print1n("dd = " + dd);
    System.out.print1n("de = " + de);
  }
}
```

Cuando se ejecuta este programa, la salida es la siguiente:

```
Aritmética de Enteros
a = 2
b = 6
c = 1
d =-1
e = 1

Aritmética de Punto Flotante
da = 2.0
db = 6.0
dc = 1.5
dd =-0.5
de = 0.5
```

El operador de módulo

El operador de módulo, %, devuelve el residuo generado por una operación de división. Se puede aplicar tanto a los tipos de punto flotante como a los tipos entero. El siguiente programa ejemplifica al operador %:

```
// Ejemplo del operador %.
class Modulo {
  public static void main(String args[]) {
    int x = 42;
    double y = 42.25;

    System.out.println("x mod 10 = " + x % 10 );
    System.out.println("y mod 10 = " + y % l0 );
  }
}
```

Al ejecutar este programa se obtiene la siguiente salida:

```
x mod 10 = 2
y mod 10 = 2.25
```

Operadores aritméticos combinados con asignación

Java proporciona operadores especiales que se pueden utilizar para combinar una operación aritmética con una asignación. Como probablemente sabe, sentencias como la siguiente son habituales en programación:

```
a = a + 4;
```

En Java, esta sentencia puede ser escrita también de la siguiente forma:

```
a += 4;
```

Esta versión utiliza el operador compuesto de asignación **+=**. Ambas sentencias llevan a cabo la misma acción: incrementan el valor de **a** en **4**.

Otro ejemplo es,

```
a = a %2;
```

que también se puede expresar como

```
a %=2;
```

En este caso, el operador **%=** sirve para obtener el residuo de la división **a**/2 y el resultado es asignado a **a**.

Existen operadores compuestos de asignación para todos los operadores aritméticos binarios. Cualquier sentencia de la forma

var = var op expresión;

se puede escribir como

var op = expresión;

Los operadores compuestos de asignación presentan dos ventajas. En primer lugar, permiten ahorrar un poco de escritura, ya que son formas abreviadas. En segundo lugar,

son implementados de forma más eficiente por el intérprete de Java que los formatos largos equivalentes. Por estas razones se emplean habitualmente en los programas profesionales escritos en Java.

El siguiente programa muestra en acción varios operadores de asignación, *op*=

```
// Ejemplo de varios operadores de asignación.
class OpIgual {
  public static void main(String args[]) {
    int a = 1;
    int b = 2;
    int c = 3;

    a += 5;
    b *= 4;
    c += a * b;
    c %= 6;
    System.out.println("a = " + a);
    System.out.println("b = " + b);
    System.out.println("c = " + c);
  }
}
```

La salida generada es la siguiente:

```
a = 6
b = 8
c = 3
```

Incremento y decremento

Los operadores de incremento y decremento de Java son respectivamente, ++ y – –. Recordará que fueron mencionados en el Capítulo 2. Ahora los analizaremos con más detalle. Estos operadores tienen ciertas propiedades especiales que los hacen muy interesantes. Comencemos revisando qué hacen exactamente estos dos operadores.

El operador de incremento aumenta en una unidad a su operando, mientras que el operador de decremento reduce en una unidad a su operando. Por ejemplo, esta sentencia:

```
x = x + 1;
```

se puede escribir también usando el operador de incremento:

```
x++;
```

Del mismo modo, esta sentencia:

```
x = x - 1;
```

es equivalente a:

```
x--;
```

Estos operadores son los únicos que pueden aparecer en forma de *sufijo*, en la que siguen al operando, como se acaba de mostrar, y en la forma de *prefijo*, en la que van delante del operando. En los siguientes ejemplos, no existe diferencia entre estas dos formas. Sin embargo,

cuando el operador incremento y/o decremento forma parte de una expresión más larga surge una ligera, aunque importante, diferencia entre estas dos formas. En la forma prefija, el operando es incrementado o decrementado antes de obtener el valor que se utilizará en la expresión. En la forma sufija, se utiliza el valor del operando en la expresión, y después se modifica. Por ejemplo:

```
x = 42;
y = ++x;
```

En este caso, se asigna a **y** el valor 43, como se podría esperar, ya que el incremento se produce *antes* de que se asigne a **y** el valor de **x**. La línea **y =++x;** es equivalente a estas otras dos sentencias:

```
x = x + 1;
y = x;
```

Sin embargo, si se escribe de esta otra forma,

```
x = 42;
y =x++;
```

primero se asigna a y el valor de **x**, y después se realiza el incremento, de manera que el valor de **y** será 42. Naturalmente, en ambos casos, el valor final de **x** es 43. La línea **y =x++;** es equivalente a estas dos sentencias:

```
y =x;
x =x + 1;
```

El siguiente programa es un ejemplo del operador incremento.

```
// Ejemplo del operador ++.
class IncDec {
  pub1ic static void main(String args[]) {
    int a = 1;
    int b = 2;
    int c;
    int d;
    c = ++b;
    d =a++;
    c++;
    System.out.println("a = " + a);
    System.out.println("b = " + b);
    System.out.println("c = " + c);
    System.out.println("d = " + d);
  }
}
```

La salida que se produce es:

```
a = 2
b = 3
c = 4
d = 1
```

Operadores a nivel de bit

Java define varios operadores a nivel de bit que se pueden aplicar a los tipos enteros, **long**, **int**, **short**, **char** y **byte**. Estos operadores actúan sobre los bits individuales de sus operandos. Estos operadores se resumen en la siguiente tabla:

Operador	Resultado
~	NOT unario a nivel de bit
&	AND a nivel de bit
|	OR a nivel de bit
^	OR exclusivo a nivel de bit
>>	Desplazamiento a la derecha
>>>	Desplazamiento a la derecha rellenando con ceros
<<	Desplazamiento a la izquierda
&=	AND a nivel de bit y asignación
|=	OR a nivel de bit y asignación
^=	OR exclusivo a nivel de bit y asignación
>>=	Desplazamiento a la derecha y asignación
>>>=	Desplazamiento a la derecha rellenando con ceros y asignación
<<=	Desplazamiento a la izquierda y asignación

Ya que los operadores a nivel de bit manipulan los bits en un entero, es importante comprender qué efectos pueden tener esas manipulaciones sobre el valor. Concretamente, es útil conocer cómo almacena Java los valores enteros y cómo representa números negativos. Así que antes de continuar, revisemos brevemente estos dos temas.

Todos los tipos enteros se representan mediante números binarios con un número distinto de bits. Por ejemplo, el valor 42, para el tipo **byte**, en binario es 00101010, donde cada posición representa una potencia de dos, comenzando con 2^0 para el bit situado más a la derecha. El siguiente bit a la izquierda sería 2^1, o 2, continuando hacia la izquierda con 2^2, o 4, y seguidamente 8, 16, 32, etc. De esta forma, 42 tiene un valor 1 en los bits de las posiciones 1, 3 y 5 (contando desde 0 a partir de la derecha); así que 42 es la suma de $2^1 + 2^3 + 2^5$, es decir, 2 + 8 + 32.

Todos los tipos enteros excepto **char** son enteros con signo. Esto significa que pueden representar tanto valores positivos como valores negativos. Java utiliza una codificación denominada *complemento a dos*, lo que significa que los números negativos se representan invirtiendo (cambiando los unos por ceros y viceversa) todos los bits en el valor y añadiendo un 1 al resultado. Por ejemplo, – 42 se representa invirtiendo todos los bits de 42, o 00101010, lo que da lugar a 11010101, y añadiendo 1 se produce el resultado final 11010110, o – 42. Para decodificar un número negativo, primero se invierten todos sus bits, y luego se suma 1. Invirtiendo por ejemplo los bits de – 42, o 11010110, se obtiene 00101001, o 41, entonces añadiendo 1 se obtiene 42.

La razón de que Java, (y la mayoría de los otros lenguajes de programación), utilice el complemento a dos es fácil de entender cuando se considera la cuestión del *cruce por cero*. Con valores del tipo **byte**, el cero se representa por 00000000. Utilizando el complemento a uno, es

decir, invirtiendo simplemente todos los bits, se obtiene 11111111, que sería la representación de un cero negativo. Pero un cero negativo no es válido en aritmética entera. Este problema se resuelve usando el complemento a dos para representar números negativos. Al usar el complemento a dos y añadir un 1 al complemento, se obtiene 100000000 como resultado. Esto produce un bit 1 por la izquierda, el noveno bit, que no cabe en un valor del tipo **byte**, y la representación que se obtiene para –0 es la misma que para 0. La representación para – 1 es 11111111. Aunque en el ejemplo anterior se ha utilizado un valor del tipo **byte**, los mismos principios básicos siguen aplicando para cualquier tipo entero en Java.

Como Java utiliza el complemento a dos para representar números negativos, y dado que todos los valores enteros en Java tienen signo —al aplicar los operadores a nivel de bit se pueden producir con facilidad resultados inesperados— Por ejemplo, si se cambia el bit de mayor orden de un valor, el resultado obtenido puede interpretarse como un número negativo, independientemente de que éste fuera el objetivo o no. Para evitar sorpresas desagradables, basta con recordar que el bit de mayor orden sólo se utiliza para determinar el signo.

Operadores lógicos a nivel de bit

Los operadores lógicos a nivel de bit son **&**, **|**, **^** y **~**. La siguiente tabla muestra el resultado de cada operación. En la siguiente discusión, hay que tener en cuenta que los operadores a nivel de bit se aplican a cada uno de los bits de los operandos.

A	B	A|B	A&B	A^B	~A
0	0	0	0	0	1
1	0	1	0	1	0
0	1	1	0	1	1
1	1	1	1	0	0

El operador NOT

El operador NOT unario, ~, también denominado *complemento* a nivel de bit, invierte todos los bits de su operando. Por ejemplo, al aplicar el operador NOT al número 42, que tiene la siguiente representación:

```
00101010
```

se obtiene

```
11010101
```

después de que el operador NOT es aplicado.

El operador AND

El operador AND, **&**, produce un bit 1 si ambos operandos son 1, y un 0 en el resto de los casos. Como ejemplo:

```
  00101010     42
& 00001111     15
----------
  00001010     10
```

El operador OR

El operador OR, |, combina los bits de manera tal que si uno de los bits del operando es un 1, entonces el bit resultante es un 1, tal y como se muestra a continuación:

```
  00101010    42
| 00001111    15
----------
  00101111    47
```

El operador XOR

El operador XOR, ^, combina los bits de tal forma que si exactamente uno de los operandos es 1, entonces el resultado es 1. En el resto de los casos el resultado es 0. El siguiente ejemplo sirve para ver el efecto del operador ^. Este ejemplo también demuestra un atributo útil de la operación XOR. Observe cómo se invierte el patrón de bits de 42 siempre que el segundo operando tenga un bit 1. Sin embargo, si el segundo operando tiene un bit 0, el primer operando no cambia. Esta propiedad es útil en determinadas manipulaciones de bits.

```
  00101010    42
^ 00001111    15
----------
  00100101    37
```

Usando los operadores lógicos a nivel de bit

El siguiente programa demuestra el funcionamiento de los operadores lógicos a nivel de bit.

```
// Ejemplo de los operadores lógicos a nivel de bit.
class BitLogic {
  public static void main(String args[]) {
    String binary[] = {
      "0000", "0001", "0010", "0011", "0100", "0101", "0110", "0111", "1000v,
      "1001", "1010", "1011", "1100", "1101", "1110", "1111"
    };
    int a = 3; // 0 + 2 + 1 ó 0011 en binario
    int b = 6; // 4 + 2 + 0 ó 0110 en binario
    int c = a | b;
    int d = a & b;
    int e = a ^ b;
    int f = (~a & b) | (a & ~b);
    int g = ~a & 0x0f ;

    System.out.println("              a = " + binary[a]);
    System.out.println("              b = " + binary[b]);
    System.out.println("          a | b = " + binary[c]);
    System.out.println("          a & b = " + binary[d]);
    System.out.println("          a ^ b = " + binary[e]);
    System.out.println("~a & b | a & ~b = " + binary[f]);
    System.out.println("             ~a = " + binary[g]);
  }
}
```

En este ejemplo, se han elegido **a** y **b** de tal forma que tienen patrones de bit que representan las cuatro combinaciones diferentes que se pueden tener con los dos dígitos binarios: 0-0, 0-1, 1-0, y 1-1. En los resultados de **c** y **d**, se puede ver cómo operan los

operadores | y & en cada bit. Los valores asignados a **e** y **f** son iguales y muestran cómo funciona el operador ^. El arreglo de tipo string llamado **binary** contiene la representación binaria de los números del 0 al 15. En este ejemplo, se indexa el arreglo para mostrar la representación binaria de cada resultado. El arreglo se ha construido de manera que la representación como cadena de un valor binario **n** esté almacenada en **binary[n].** Al valor de **~a** se le hace un AND con **0x0f** (0000 1111 en binario) para reducir su valor a un valor menor a 16, y poder imprimirlo utilizando el arreglo **binary**. La salida de este programa es:

```
          a = 0011
          b = 0110
      a | b = 0111
      a & b = 0010
      a ^ b = 0101
  ~a&b|a&~b = 0101
         ~a = 1100
```

Desplazamiento a la izquierda

El operador de desplazamiento a la izquierda, **<<**, desplaza a la izquierda todos los bits de un valor un determinado número de veces. Su forma general es:

valor << *num*

num especifica el número de posiciones que se desplazarán a la izquierda los bits de *valor*; es decir, el operador << mueve todos los bits del valor especificado a la izquierda el número de posiciones indicado por *num*. En cada desplazamiento a la izquierda el bit de mayor orden es desplazado fuera (y perdido), y un cero incluido a la derecha. Esto significa que cuando se realiza un desplazamiento a la izquierda en un operando del tipo **int**, se pierden los bits una vez que son desplazados más allá de la posición 31. Si el operando es del tipo **long**, entonces los bits se pierden después de la posición 63.

La promoción automática de tipos de Java da lugar a resultados inesperados cuando se desplazan valores del tipo **byte** y **short**. Los valores del tipo **byte** y **short** son promocionados a **int** cuando se evalúa una expresión. Además, el resultado de la misma expresión también es del tipo **int**. Esto significa que el resultado de un desplazamiento a la izquierda en un valor del tipo **byte** o **short** dará lugar a un valor **int**, y los bits desplazados a la izquierda no se perderán mientras no vayan más allá de la posición 31. Asimismo, en un valor **byte** o **short** negativo el signo se extiende cuando es promocionado a **int**, y los bits de mayor orden se rellenan con unos. Por esta razón, al realizar un desplazamiento a la izquierda en un valor **byte** o **short** se debe descartar los bits de mayor orden del resultado **int**. Por ejemplo, si se desplaza a la izquierda un valor byte, ese valor será promocionado a **int** y después desplazado. Esto implica que se deben descartar los tres bits más altos del resultado si lo que se quiere es el valor **byte** desplazado. La manera más fácil de hacer esto es, simplemente, convertir el resultado al tipo **byte**. El siguiente programa muestra este concepto:

```
// Desplazamiento a la izquierda en un valor byte.
class DesplazarByte {
  public static void main(String args([]) {
    byte a = 64, b;
    int i;
    i = a << 2;
    b = (byte) (a << 2);
```

```
    System.out.println("Valor original de a: " + a);
    System.out.println("i y b: " + i + " " + b);
  }
}
```

La salida que produce este programa es:

```
Valor original de a: 64
i y b: 256 0
```

El valor de **a** es promocionado a **int** en la evaluación, y el desplazamiento hacia la izquierda del valor 64 (0100 0000) dos veces da como resultado el valor 256 (1 0000 0000), que es almacenado en **i**. Sin embargo, el valor contenido en **b** es 0 ya que, después del desplazamiento, el bit de menor orden es 0, y su único 1 ha sido desplazado fuera.

Como el desplazamiento a la izquierda tiene el efecto de multiplicar por dos el valor original, los programadores lo utilizan como una alternativa eficiente para realizar dicha multiplicación, pero teniendo en cuenta que al desplazar un 1 en el bit de mayor orden (bit 31 o 63), el valor que se obtiene es negativo. Esto se ejemplifica en el siguiente programa.

```
// El desplazamiento a la izquierda es una forma rápida de multiplicar por 2.
class MultPorDos {
  public static void main(String args([]) {
    int i;
    int num = 0xFFFFFFE;

    for (i=0; i<4; i++) {
      num = num << 1;
      System.out.println(num);
    }
  }
}
```

La salida que se obtiene es:

```
536870908
1073741816
2147483632
-32
```

El valor de partida fue elegido cuidadosamente para que después de desplazarlo cuatro posiciones diera lugar a –32. Como puede verse, cuando se desplaza un bit 1 a la posición 31, el número se interpreta como negativo.

Desplazamiento a la derecha

El operador de desplazamiento a la derecha, **>>**, desplaza todos los bits de un valor a la derecha un número especificado de veces. Su forma general es:

valor >> *num*

num especifica el número de posiciones que se desplazarán a la derecha los bits de *valor*; es decir, el operador **>>** mueve todos los bits del valor especificado el número de veces que indica *num*.

El siguiente fragmento de código desplaza dos posiciones a la derecha el valor 32, lo que da por resultado un 8 que se almacena en la variable a:

```
int a = 32;
a = a >> 2; // ahora a contiene 8
```

Cuando un valor tiene bits que son desplazados fuera del rango de posiciones, esos bits se pierden. Por ejemplo, en el siguiente fragmento de código se desplaza el valor 35 a la derecha dos posiciones. Esto hace que los dos bits de orden más bajo se pierdan y el resultado final en **a** sea 8.

```
int a = 35;
a = a >> 2; / / ahora también se obtiene un 8 en a
```

Revisando la operación en binario, se ve más claramente lo que pasa:

```
00100011    35
>>2
00001000    8
```

Cada vez que se desplaza un valor a la derecha, se divide ese valor por dos y se pierde el residuo. Esto se puede utilizar para realizar divisiones enteras entre dos con un mayor rendimiento. Obviamente, hay que asegurar que no se pierdan bits por la derecha.

Cuando se realiza el desplazamiento a la derecha, los bits superiores (más a la izquierda) se rellenan con el contenido previo del bit superior. A esto se denomina extensión de signo, y sirve para preservar el signo de los números negativos cuando se realizan desplazamientos a la derecha. Por ejemplo, –8 >> 1 es igual a –4, que expresado en binario es:

```
11111000    –8
>>1
11111100    –4
```

Es interesante observar que si se desplaza a la derecha –1, el resultado sigue siendo –1, ya que la extensión de signo introduce un 1 en el bit de mayor orden.

En ocasiones, puede que no se quiera realizar la extensión de signo al realizar desplazamientos a la derecha. El siguiente programa, por ejemplo, convierte un valor **byte** en su representación hexadecimal tipo cadena. Observe que el valor desplazado es enmascarado cuando se le aplica el operador AND con **0x0f** para descartar cualquier bit de extensión de signo, y este valor se puede utilizar como índice en la matriz de caracteres hexadecimales

```
// Enmascarando la extensión de signo.
class HexByte (
  static public void main(String args[]) {
    char hex[] = {
      '0', '1', '2', '3', '4', '5', '6', '7',
      '8', '9', 'a', 'b', 'c', 'd', 'e', 'f'
    };

    byte b = (byte) 0xf1;

    System.out.println("b = 0x" + hex [ (b >> 4) & 0x0f] + hex [b & 0x0f]);
  }
}
```

La salida de este programa es:

```
  b = 0xf1
```

Desplazamiento a la derecha sin signo

Como hemos visto, el operador **>>** rellena automáticamente el bit de mayor orden con su contenido previo cada vez que se realiza un desplazamiento, esto preserva el signo del valor. Sin embargo, puede ser que algunas veces no se desee hacer esto; por ejemplo, si se desplaza algo que no representa un valor numérico, entonces podría no desear la preservación del signo. Esta situación es frecuente cuando se trabaja con valores basados en pixeles y gráficos. En este caso interesa desplazar un cero en el bit de mayor orden, independientemente de cual sea su valor inicial. Este desplazamiento se denomina *desplazamiento sin signo*. Para realizar esta operación en Java se utiliza el operador de desplazamiento a la derecha sin signo, **>>>**, que siempre desplaza ceros en el bit de mayor orden.

El siguiente fragmento de código muestra cómo funciona el operador **>>>**. Se asigna a **a** el valor –1, que tiene una representación binaria con 32 unos. A continuación se desplaza a la derecha 24 posiciones, rellenando los 24 bits más altos con ceros, ignorando la extensión de signo normal. Esto hace que el valor final de **a** sea 255.

```
int a = -1;
a = a >>> 24;
```

Revisemos la misma operación en binario para ilustrar mejor como tiene lugar el desplazamiento:

```
11111111 11111111 11111111 11111111      –1 en binario como un int
>>> 24
00000000 00000000 00000000 11111111      255 en binario como un int
```

A menudo, el operador **>>>** no es todo lo útil que puede parecer en un principio, ya que sólo afecta a valores de 32 y 64 bits. Recuerde que los valores más pequeños son promocionados automáticamente al tipo **int**. Esto significa que la extensión de signo y el desplazamiento tienen lugar en un valor de 32 bits, en lugar de valores de 8 o 16 bits. Se puede suponer que un desplazamiento a la derecha sin signo es un valor del tipo **byte** relleno con ceros a partir del bit séptimo, pero no es así, ya que realmente es un valor de 32 bits el que se desplaza. El siguiente programa ejemplifica este efecto:

```
// Desplazamiento sin signo en un valor byte.
class ByteUShift {
  static public void main(String args[]) {
    char hex[] = {
      '0', '1', '2', '3', '4', '5', '6', '7',
      '8', '9', 'a', 'b', 'c', 'd', 'e', 'f'
    };
    byte b = (byte) 0xf1;
    byte c = (byte) (b >> 4);
    byte d = (byte) (b >>> 4);
    byte e = (byte) ((b & 0xff) >> 4);
    System.out.println("              b = 0x"
      + hex[(b >> 4) & 0x0f] + hex[b & 0x0f]);
    System.out.println("         b >> 4 = 0x"
      + hex[(c >> 4) & 0x0f] + hex[c & 0x0f]);
```

```
    System.out.println("        b >>> 4 = 0x"
      + hex[(d >> 4) & 0x0f] + hex[d & 0x0f]);
    System.out.println("(b & 0xff) >> 4 = 0x"
      + hex[(e >> 4) & 0x0f] + hex[e & 0x0f]);
  }
}
```

La salida de este programa muestra cómo, aparentemente, el operador **>>>** no tiene ningún efecto cuando opera sobre un valor byte. Para demostrar esto, se asigna arbitrariamente a la variable **b** un valor negativo del tipo **byte**. A continuación se asigna a **c** el valor **byte** que resulta al desplazar a la derecha cuatro posiciones **b**, que es 0xff teniendo en cuenta la esperada extensión de signo. Después se asigna a **d** el valor **byte** de **b** desplazado a la derecha sin signo cuatro posiciones, que se podría suponer igual a 0x0f, pero que realmente es 0xff debido a la extensión de signo que se produce cuando se promociona **b** a un valor **int** antes del desplazamiento. La última expresión asigna a **e** el valor **byte** de **b** enmascarado a 8 bits usando el operador AND, y desplazando a la derecha cuatro posiciones, que da lugar al valor esperado de 0x0f. Observe que para obtener **d** no se utiliza el operador de desplazamiento a la derecha sin signo, ya que se conoce el estado del bit de signo después de AND.

```
                b = 0xf1
           b >> 4 = 0xff
          b >>> 4 = 0xff
(b & 0xff) >> 4 = 0x0f
```

Operadores a nivel de bit combinados con asignación

Todos los operadores a nivel de bit binarios tienen una forma abreviada similar a la de los operadores algebraicos, que combina la asignación con la operación a nivel de bit. Por ejemplo, las dos sentencias siguientes, que desplazan el valor de **a** a la derecha en cuatro bits, son equivalentes:

```
a = a >> 4;
a >>= 4;
```

Del mismo modo, mediante las dos sentencias siguientes, que son equivalentes, se asigna a **a** la expresión **a** OR **b** a nivel de bit.

```
a = a | b;
a |= b;
```

El siguiente programa crea algunas variables enteras y utiliza la forma abreviada de los operadores de asignación a nivel de bit para manipularlas:

```
class OpBitEquals {
  public static void main(Stringargs[]) {
    int a = 1;
    int b = 2;
    int c = 3;

    a |= 4;
    b >>= 1;
    c <<= 1;
    a ^= c;
```

```
    System.out.println("a = " + a);
    System.out.println("b = " + b);
    System.out.println("c = " + c);
  }
}
```

La salida de este programa es la siguiente:

```
a = 3
b = 1
c = 6
```

Operadores relacionales

Los *operadores relacionales* determinan la relación que un operando tiene con otro. Específicamente, determinan relaciones de igualdad y orden. A continuación se muestran los operadores relacionales:

Operador	Resultado
==	Igual a
!=	Diferente de
>	Mayor que
<	Menor que
>=	Mayor o igual que
<=	Menor o igual que

El resultado de estas operaciones es un valor **booleano**. La aplicación más frecuente de los operadores relacionales es en la obtención de expresiones que controlan la sentencia **if** y las sentencias de ciclos.

En Java se pueden comparar valores de cualquier tipo, incluyendo los números enteros y de punto flotante, los caracteres y valores **booleanos**, usando la prueba de igualdad, **==**, y la de desigualdad, **!=**. Observe que, la igualdad se indica utilizando dos signos igual, y no sólo uno. (Recuerde que un signo igual es el operador de asignación). Sólo se pueden comparar valores numéricos mediante los operadores de orden, es decir, solamente se pueden comparar operandos enteros, de punto flotante y caracteres, para ver cuál es mayor o menor.

Como se mencionó, el resultado que produce un operador relacional es un valor de tipo **boolean**. El siguiente fragmento de código, por ejemplo, es perfectamente válido:

```
int a = 4;
int b = 1;
boolean c = a < b;
```

En este caso, se almacena en **c** el resultado de **a < b** (que es la literal **false**).

Si usted tiene experiencia programando en C/C++ considere lo siguiente, estos dos tipos de sentencias son muy comunes:

```
int done;
/ / ...
```

```
if(!done) …//Válido en C/C++
if(done) ... //pero no en Java
```

En Java, estas dos sentencias deben ser escritas en la forma siguiente:

```
if (done == 0) ... // Este es el estilo de Java.
if (done != 0) ...
```

La diferencia está en que Java no define los valores verdadero y falso del mismo modo que en C/C++, donde cualquier valor distinto de cero es verdadero, y el cero es falso. En Java los valores **true** y **false** son valores no numéricos que no tienen relación con el cero o el distinto de cero. Por esto, para comprobar si un valor es igual a cero o no, se deben emplear explícitamente uno o más operadores relacionales.

Operadores lógicos booleanos

Los operadores lógicos **booleanos** que se muestran a continuación sólo operan sobre operandos del tipo **boolean**. Todos los operadores lógicos binarios combinan dos valores boolean para dar como resultado un valor **boolean**.

Operador	Resultado
&	AND lógico
|	OR lógico
^	XOR lógico (OR exclusivo)
||	OR en cortocircuito
&&	AND en cortocircuito
!	NOT lógico unario
&=	Asignación AND
|=	Asignación OR
^=	Asignación XOR
==	Igual a
!=	Diferente de
?:	if-then-else ternario

Los operadores lógicos **booleanos**, **&, | ,** y **^,** operan sobre valores del tipo **boolean** de la misma forma que operan sobre los bits de un entero. El operador lógico **!**, invierte el estado **booleano**: **!true == false** y **!false== true**. La siguiente tabla muestra el resultado de cada operación lógica:

A	B	A|B	A&B	A^B	!A
false	false	false	false	false	true
true	false	true	false	true	false
false	true	true	false	true	true
true	true	true	true	false	false

El siguiente programa es casi igual que el ejemplo **BitLogic** mostrado anteriormente, pero este utiliza valores lógicos del tipo **boolean** en lugar de bits:

```
// Ejemplo de los operadores lógicos booleanos.
class BoolLogic (
  public static void main(String args[]) {
    boolean a = true;
    boolean b = false;
    boolean c = a | b;
    boolean d = a & b;
    boolean e = a ^ b;
    boolean f = (!a & b) | (a & !b);
    boolean g = !a;
    System.out.println("               a = " + a);
    System.out.println("               b = " + b);
    System.out.println("           a | b = " + c);
    System.out.println("           a & b = " + d);
    System.out.println("           a ^ b = " + e);
    System.out.println(" !a & b | a & !b = " + f);
    System.out.println("              !a = " + g);
  }
}
```

Después de ejecutar este programa, se ve que las reglas lógicas se aplican a los valores de tipo **boolean** del mismo modo que se aplicaron a los bits. La siguiente salida muestra que la representación como cadena de los valores **boolean** en Java son las literales, **true** o **false**:

```
               a = true
               b = false
           a | b = true
           a & b = false
           a ^ b = true
  a & b | a & !b = true
              !a = false
```

Operadores lógicos en cortocircuito

Java proporciona dos operadores lógicos **booleanos** que no se encuentran en muchos otros lenguajes de programación. Se trata de versiones secundarias de los operadores AND y OR, y se conocen como operadores lógicos en *cortocircuito*. En la tabla anterior se ve cómo el operador OR da como resultado **true** cuando **A** es **true**, independientemente del valor de **B**. Del mismo modo, el operador AND da como resultado **false** cuando **A** es **false**, independientemente del valor de **B**. Cuando se utilizan las formas **||** y **&&**, en lugar de las formas **|** y **&** de estos operadores, Java no evalúa el operando de la derecha si el resultado de la operación queda determinado por el operando de la izquierda. Esto es útil cuando el operando de la derecha depende de que el de la izquierda sea true o false. El siguiente fragmento de código, por ejemplo, muestra cómo se puede utilizar de manera ventajosa la evaluación lógica en cortocircuito para asegurar que la operación de división sea válida antes de evaluarla:

```
if (denom != 0 && num / denom > 10)
```

Al usar la forma en cortocircuito del operador AND (**&&**) no existe riesgo de que se produzca una excepción en tiempo de ejecución si **denom** es igual a cero. Si esta línea de

código se escribiera utilizando la versión sencilla del operador AND, **&**, se deberían evaluar ambos operandos, y se produciría una excepción cuando el valor de **denom** fuera igual a cero.

Es una práctica habitual el usar las formas en cortocircuito de los operadores AND y OR en los casos de lógica booleana, y dejar la versión de un sólo carácter de estos operadores exclusivamente para las operaciones a nivel de bit. Sin embargo, hay excepciones a esta regla. Consideremos, por ejemplo, la siguiente sentencia:

```
if (c == l & e++ < 100) d = 100;
```

Aquí, al utilizar un solo carácter, **&**, se asegura que la operación de incremento se aplicará a **e** tanto si **c** es igual a 1 como si no lo es.

El operador de asignación

Aunque el operador de asignación se ha estado utilizando desde el Capítulo 2, en este momento se puede analizar de manera más formal. El *operador de asignación* es un solo signo igual, **=**. Este operador se comporta en Java del mismo modo que en otros lenguajes de programación. Tiene la forma general:

var = *expresión*;

donde el tipo de la variable **var** debe ser compatible con el tipo de expresión.

El operador de asignación tiene un atributo interesante con el que puede que no esté familiarizado: permite crear una cadena de asignaciones. Consideremos, por ejemplo, este fragmento de código:

```
int x, y, z;

x = y = z = 100; // asigna a x, y, y z el valor 100
```

Este código, asigna a las variables **x**, **y** y **z** el valor 100 mediante una única sentencia. Y esto es así porque el operador **=** es un operador que cede el valor de la expresión de la derecha. Por tanto, el valor de **z =100** es 100, que entonces se asigna a **y**, y que a su vez se asigna a **x**. Utilizar una "cadena de asignaciones" es una forma fácil de asignar a un grupo de variables un valor común.

El operador ?

Java incluye un *operador ternario* especial que puede sustituir a ciertos tipos de sentencias if-then-else. Este operador es **?**. Puede resultar un tanto confuso en principio, pero el operador **?** resulta muy efectivo una vez que se ha practicado con él. El operador **?** tiene la siguiente forma general:

expresión1 **?** *expresión2* : *expresión3*

Donde *expresión1* puede ser cualquier expresión que dé como resultado un valor del tipo **boolean**. Si *expresión1* genera como resultado **true**, entonces se evalúa la *expresión2*; en caso contrario se evalúa la *expresión3*. El resultado de la operación **?** es el de la expresión evaluada. Es necesario que tanto la *expresión2* como la *expresión3* devuelvan el mismo tipo que no puede ser **void**.

A continuación se presenta un ejemplo de empleo del operador **?**:

```
resultado = denom == 0 ? 0 : num / denom;
```

Cuando Java evalúa esta expresión de asignación, en primer lugar examina la expresión a la *izquierda* de la interrogación. Si **denom** es igual a cero, se evalúa la expresión que se encuentra *entre* la interrogación y los dos puntos y se toma como valor de la expresión completa. Si **denom** no es igual a cero, se evalúa la expresión que está detrás de los dos puntos y se toma como valor de la expresión completa. Finalmente, el resultado del operador **?** se asigna a la variable **resultado**.

El siguiente programa es un ejemplo del operador **?**, que se utiliza para obtener el valor absoluto de una variable.

```
// Ejemplo del operador ?
class Ternario {
  public static void main(String args[]) {
    int i. k;

    i = 10;
    k = i < 0 ? -i : i; // se obtiene el valor absoluto de i
    System.out.print("Valor absoluto de ");
    System.out.println(i + " es " + k);

    i = -10;
    k = i < 0 ? -i : i; // se obtiene el valor absoluto de i
    System.out.print("Valor absoluto de" );
    System.out.println(i + " es " + k);
  }
}
```

La salida que se obtiene es:

```
El valor absoluto de 10 es 10
El valor absoluto de -10 es 10
```

Precedencia de operadores

La Tabla 4.1 muestra el orden de precedencia de los operadores de Java, desde la más alta a la más baja. Observe que la primera fila presenta elementos a los que normalmente no se considera como operadores: paréntesis, corchetes y el operador punto. Técnicamente, éstos son llamados *separadores*, pero ellos actúan como operadores en una expresión. Los paréntesis son usados para alterar la precedencia de una operación. Después de haber visto los capítulos anteriores, ya sabemos que los corchetes se utilizan para indexar arreglos. El operador punto se utiliza para acceder a los elementos contenidos en un objeto y se discutirá más adelante.

El uso de paréntesis

Los *paréntesis* aumentan la prioridad de las operaciones en su interior. Esto es necesario para obtener el resultado deseado en muchas ocasiones. Consideremos la siguiente expresión:

```
a >> b + 3
```

En esta expresión, en primer lugar se añaden 3 unidades a **b** y después se desplaza a **a** la derecha tantas posiciones como el resultado de la suma anterior. Esta expresión se puede escribir también utilizando paréntesis:

```
a >> (b + 3)
```

TABLA 4.1 Precedencia de los Operadores en Java

Precedencia más alta			
()	[]	.	
++	– –	~	!
*	/	%	
+	–		
>>	>>>	<<	
>	>=	<	<=
==	!=		
&			
^			
|			
&&			
||			
?:			
=	op=		

Sin embargo, si en primer lugar, se quiere desplazar **a** a la derecha las posiciones que indique **b**, y después añadir 3 al resultado, será necesario utilizar paréntesis.

```
(a >> b) + 3
```

Además de cambiar la prioridad de un operador, los paréntesis se utilizan en algunas ocasiones para hacer más claro el significado de una expresión. Para cualquiera que lea su código, una expresión compleja puede ser difícil de entender. Añadir paréntesis puede ser redundante, pero ayuda a que expresiones complejas resulten más claras, evitando posibles confusiones posteriores. Por ejemplo, ¿cuál de las siguientes expresiones es más fácil de leer?

```
a | 4 + c >> b & 7
(a | ( ( (4 + c) >> b) & 7) )
```

Una cuestión más: los paréntesis, redundantes o no, no degradan el funcionamiento de un programa. Por lo tanto, añadir paréntesis para reducir ambigüedades no afecta negativamente al programa.

CAPÍTULO 5

Sentencias de control

Un lenguaje de programación utiliza sentencias de *control* para hacer que el flujo de ejecución avance y se bifurque en función de los cambios de estado en el programa. Las sentencias de control para programas en Java pueden ser clasificadas en las siguientes categorías: selección, iteración y salto. Las sentencias de *selección* permiten al programa elegir diferentes caminos de ejecución con base en el resultado de una expresión o en el estado de una variable. Las sentencias de *iteración* permiten al programa ejecutar repetidas veces una o más sentencias (las sentencias de iteración constituyen los ciclos). Finalmente, las sentencias de *salto* hacen posible que el programa se ejecute de una forma no lineal. En este capítulo se examinan las sentencias de control de Java.

Sentencias de selección

Java admite dos sentencias de selección: **if** y **switch**. Estas sentencias permiten controlar el flujo de ejecución del programa basado en función de condiciones conocidas únicamente durante el tiempo de ejecución. Se sorprenderá gratamente de la potencia y flexibilidad de estas dos sentencias.

if

La sentencia **if** se introdujo en el Capítulo 2 y se examina con detalle en este capítulo, **if** es la sentencia de bifurcación condicional de Java. Se puede utilizar para dirigir la ejecución del programa hacia dos caminos diferentes. El formato general de la sentencia **if** es:

if *(condición) sentencia1;*
else *sentencia2;*

Cada sentencia puede ser una sentencia única o un conjunto de sentencias encerradas entre llaves, es decir, un bloque. La condición es cualquier expresión que devuelva un valor **booleano**. La cláusula **else** es opcional.

La sentencia **if** funciona del siguiente modo: Si la *condición* es verdadera, se ejecuta la *sentencia1*. En caso contrario se ejecuta la *sentencia2* (si es que existe). En ningún caso se ejecutarán ambas sentencias. Las siguientes líneas muestran un ejemplo en el que se utiliza la sentencia if.

```
int a, b;
// ...
if (a < b) a = 0;
else b = 0;
```

Si **a** es menor que **b**, entonces **a** se hace igual a cero. En caso contrario, **b** se hace igual a cero. En ningún caso se asignará a ambas variables el valor cero.

Con mucha frecuencia, la expresión que se utiliza para controlar la sentencia **if** involucrará operadores relacionales. Sin embargo, esto no es técnicamente necesario. Es posible controlar la sentencia **if** utilizando una sola variable **booleana** como se muestra en el siguiente fragmento de código:

```
boolean datosDisponibles;
// ...
if (datosDisponibles)
  procesarDatos();
else
  esperarDatos () ;
```

Recuerde que sólo una sentencia puede aparecer inmediatamente después del **if** o del **else**. Si se quiere incluir más sentencias, es necesario crear un bloque tal y como se hace a continuación:

```
int bytesDisponibles;
// ...
if (bytesDisponibles > 0) {
procesarDatos () ;
bytesDisponibles -= n;
} else
esperarDatos ( ) ;
```

Aquí, las dos sentencias contenidas en el bloque **if** serán ejecutadas si **bytesDisponibles** es mayor que cero.

Algunos programadores estiman conveniente utilizar las llaves siempre que utilizan la sentencia **if**, incluso aunque sólo haya una sentencia en la cláusula. Esto facilita añadir otras sentencias en un momento posterior y no hay que preocuparse por haber olvidado las llaves. De hecho, una causa bastante común de errores es olvidar definir un bloque cuando es necesario. En el siguiente fragmento de código se muestra un ejemplo:

```
int bytesDisponibles;
// ...
if (bytesDisponibles > 0){
  procesarDatos();
  bytesDisponibles -= n;
}else
  esperarDatos();
  bytesDisponibles = n;
```

Parece evidente que la sentencia **bytesDisponibles = n**; debía haber sido ejecutada dentro de la cláusula **else** teniendo en cuenta su nivel de identación. Sin embargo, como recordará, un espacio en blanco es insignificante para Java y no es posible que el compilador reconozca qué se quería hacer en realidad. Este código compilará correctamente pero se comportará de manera errónea cuando se ejecute.

El ejemplo anterior se corrige en el código que sigue a continuación:

```
int bytesDisponibles;
// ...
```

```
if (bytesDisponibles > 0) {
  procesarDatos();
  bytesDisponibles -= n;
} else {
  esperarDatos();
  bytesDisponibles = n;
}
```

if anidados

Un **if** *anidado* es una sentencia **if** que está contenida dentro de otro **if** o **else**. Los **if** anidados son muy habituales en programación. Cuando se anidan **if** lo más importante es recordar que una sentencia **else** siempre corresponde a la sentencia **if** más próxima dentro del mismo bloque y que no esté ya asociada con otro **else**. Veamos un ejemplo:

```
if (i == 10) {
  if (j < 20) a = b;
  if (k > 100) c = d;  // este if está
  else a =c;           // asociado con este else
}
else a = d;            // este else se refiere a if (i == 10)
```

Tal como indican los comentarios, el **else** final no está asociado con **if (j < 20)**, ya que no están dentro del mismo bloque (aunque se trate del **if** más próximo sin un **else**). La sentencia else final está asociada con **if (i == 10).** El **else** interior corresponde al **if (k > l00)**, ya que éste es el **if** más próximo dentro del mismo bloque.

if-else-if múltiples

Una construcción muy habitual en programación es la de *if-else-if múltiples*. Esta construcción se basa en una secuencia de if anidados. Su formato es el siguiente:

```
if (condición)
  sentencia;
else if (condición)
  sentencia;
else if (condición)
  sentencia;
.
.
.
else
  sentencia;
```

La sentencia **if** se ejecuta de arriba abajo. Tan pronto como una de las condiciones que controlan el **if** sea **true**, las sentencias asociadas con ese **if** serán ejecutadas, y el resto ignoradas. Si ninguna de las condiciones es verdadera, entonces se ejecutará el **else** final. El **else** final actúa como una condición por omisión, es decir, si todas las demás pruebas condicionales fallan, entonces se ejecutará la sentencia del último **else**.

Si no hubiera un **else** final y todas las demás condiciones fueran **false**, entonces no se ejecutará ninguna acción.

El siguiente programa utiliza un **if-else-if** múltiple para determinar en qué estación se encuentra un mes particular.

```
// Ejemplo de sentencias if-e1se-if.
c1ass IfElse {
  pub1ic static void main (String args[]) {
    int mes = 4;  // Abril
    String estacion;

    if (mes == 12 || mes == 1 || mes == 2)
      estacion = "Invierno";
    e1se if (mes == 3 || mes == 4 || mes == 5)
      estacion = "Primavera";
    e1se if (mes == 6 || mes == 7 || mes == 8)
      estacion = "Verano";
    e1se if (mes == 9 || mes == 10 || mes == 11)
      estacion = "Otoño";
    else
      estacion = "Mes desconocido";

    System.out.println ("Abril está en " + estación + ".");
  }
}
```

Ésta es la salida que se obtiene al ejecutar este programa:

```
Abril está en Primavera.
```

Analicemos este programa antes de continuar. Se puede comprobar que independientemente del valor de **mes**, sólo se ejecutará una sentencia de asignación.

switch

La sentencia **switch** es una sentencia de bifurcación múltiple de Java. Esta sentencia proporciona una forma sencilla de dirigir la ejecución a diferentes partes del programa en función del valor de una expresión. Así, en muchas ocasiones, es una mejor alternativa que una larga serie de sentencias **if-else-if**. El formato general de una sentencia **switch** es:

```
switch (expresión) {
  case valorl:
    // secuencia de sentencias
    break;
  case valor2:
    // secuencia de sentencias
    break;
.
.
.
case valorN:
    // secuencia de sentencias
    break;
```

```
    default:
      // secuencia de sentencias por omisión
}
```

La *expresión* debe ser del tipo **byte**, **short**, **int** o **char**; cada uno de los *valores* especificados en las sentencias **case** debe ser de un tipo compatible con el de la expresión. (También puede utilizar una enumeración para controlar un sentencia **switch**. Las enumeraciones son descritas en el Capítulo 12). Cada uno de estos valores debe ser un literal único, es decir, una constante no una variable. No se permite que aparezcan valores duplicados en las sentencias **case**.

La sentencia **switch** funciona de la siguiente forma: se compara el valor de la expresión con cada uno de los valores constantes que aparecen en las sentencias **case**. Si coincide con alguno, se ejecuta el código que sigue a la sentencia **case**. Si ninguna de las constantes coincide con el valor de la expresión, entonces se ejecuta la sentencia **default**. Sin embargo, la sentencia **default** es opcional. Si ningún **case** coincide y no existe la sentencia **default**, no se ejecuta ninguna acción.

La sentencia **break** se utiliza dentro del **switch** para terminar una secuencia de sentencias. Cuando aparece una sentencia **break**, la ejecución del código se bifurca hasta la primera línea que se encuentra después de la sentencia **switch**. El efecto que se consigue es el de"saltar fuera" del **switch**.

A continuación se presenta un ejemplo sencillo de la sentencia **switch**:

```
// Un ejemplo sencillo de switch.
class EjemploSwitch {
  public static void main (String args[]) {
    for (int i=0; i<6; i++)
      switch (i) {
        case 0:
          System.out.println ("i es cero.");
          break;
        case 1:
          System.out.println ("i es uno.");
          break;
        case 2:
          System.out.println ("i es dos.");
          break;
        case 3:
          System.out.println ("i es tres");
          break;
        default:
          System.out.println ("i es mayor que 3.");
      }
  }
}
```

La salida que tiene lugar cuando se ejecuta este fragmento de código es la siguiente:

```
i es cero.
i es uno.
i es dos.
i es tres.
i es mayor que 3.
i es mayor que 3.
```

Como se puede ver, cada vez que se ejecuta el ciclo se ejecutan las sentencias asociadas con la constante **case** que coincide con **i**. Todas las demás son ignoradas. Cuando **i** es mayor que 3, ninguna constante **case** coincide, y se ejecuta la sentencia **default**.

La sentencia **break** es opcional. Si se omite **break**, la ejecución continúa hasta el siguiente **case**. A veces, es conveniente tener múltiples sentencias **case** sin ninguna sentencia **break** entre ellas. Por ejemplo, considere el siguiente programa:

```
// En un switch, las sentencias break son opcionales.
class BreakAusente {
  public static void main (String args[]) {
    for (int i=0; i<12; i++)
      switch (i) {
        case 0:
        case 1:
        case 2:
        case 3:
        case 4:
          System.out.println ("i es menor que 5");
          break;
        case 5:
        case 6:
        case 7:
        case 8:
        case 9:
          System.out.println ("i es menor que 10");
          break;
        default:
          System.out.println ("i es 10 o mayor");
      }
  }
}
```

Este programa genera la siguiente salida:

```
i es menor que 5
i es menor que 5
i es menor que 5
i es menor que 5
i es menor que 5
i es menor que 10
i es menor que 10
i es menor que 10
i es menor que 10
i es menor que 10
i es 10 o mayor
i es 10 o mayor
```

Como puede verse, la ejecución pasa a través de cada sentencia **case** hasta que se llega a un **break**, o al final del **switch**.

Evidentemente el código anterior no es más que un ejemplo ideado para aclarar la forma en que funciona la sentencia **switch**. Sin embargo, omitir la sentencia **break** tiene muchas

aplicaciones prácticas en programas reales. Para mostrar un uso más realista considere esta otra versión del ejemplo de las estaciones presentado anteriormente. Esta versión utiliza un **switch** para conseguir una implementación más eficiente.

```
// Una versión mejorada del programa de las estaciones.
class Switch {
    public static void main (String args[]) {
        int mes = 4;
        String estacion;
        switch (mes) {
          case 12:
          case 1:
          case 2:
            estacion = "Invierno";
            break;
          case 3:
          case 4:
          case 5:
            estacion = "Primavera";
            break;
          case 6:
          case 7:
          case 8:
            estacion = "Verano";
            break;
          case 9:
          case 10:
          case 11:
            estacion = "Otoño";
            break;
          default:
            estacion = "Mes desconocido";
        }
        System.out.println ("Abri1 está en " + estacion + ".");
    }
}
```

Sentencias switch anidadas

Se puede utilizar un **switch** como parte de la secuencia de sentencias de un **switch** exterior. A esto se denomina **switch** *anidado*. Dado que una sentencia **switch** define su propio bloque, no surgen conflictos entre el **case** contenido en el **switch** interior y los contenidos en el **switch** exterior. Por ejemplo, el siguiente fragmento de código es perfectamente válido:

```
switch (contador) {
  case 1:
    switch (var) {  // switch anidado
      case 0:
        System.out.println ("var es cero");
        break;
      case 1: // no hay conflictos con el switch exterior
        System.out.println ("var es uno");
        break;
```

```
    }
    break;
case 2: // …
```

Aquí la sentencia **case 1** del **switch** interior no entra en conflicto con la sentencia **case 1** del **switch** exterior. La variable **contador** sólo se compara con la lista de las constantes **case** del nivel exterior. Si **contador** es igual a l, entonces la variable **var** se compara con la lista de constantes case del **switch** interior.

En resumen, la sentencia **switch** tiene tres características destacables:

- La sentencia **switch** se diferencía de la sentencia **if** en que la primera sólo comprueba la igualdad, mientras que la segunda puede evaluar cualquier tipo de expresión **boolenana**. Es decir, la sentencia **switch** busca solamente la coincidencia entre el valor de la expresión y una de las constantes de las sentencias **case.**
- Dos constantes de dos sentencias **case** en un mismo **switch** no pueden tener el mismo valor. Sin embargo, sí puede ocurrir que una sentencia **switch** contenida dentro de otro **switch** exterior tengan constantes iguales en sus correspondientes sentencias **case**.
- Una sentencia **switch** es más eficiente que un conjunto de sentencias **if** anidadas.

El último punto es especialmente interesante, ya que da una idea de cómo funciona el compilador Java. Cuando se compila una sentencia **switch**, el compilador Java examina cada una de las constantes en las sentencias **case** y crea una"tabla de salto"que se utiliza para seleccionar el camino de ejecución en función del valor de la expresión. Por ello, en caso de que sea necesario seleccionar entre un gran grupo de valores, la sentencia **switch** se ejecutará mucho más rápidamente que el código equivalente formado con una sucesión de sentencias **if-else**. El compilador puede hacer esto, ya que sabe que las constantes de las sentencias **case** son todas del mismo tipo y simplemente deben ser comparadas con el valor de la expresión de la sentencia **switch**. El compilador no tiene el mismo conocimiento acerca de una larga lista de expresiones **if**.

Sentencias de iteración

Las sentencias de iteración de Java son **for**, **while** y **do-while**. Estas sentencias crean lo que comúnmente se denominan *ciclos*. Como probablemente sabe, un ciclo ejecuta repetidas veces el mismo conjunto de instrucciones hasta que se cumple una determinada condición. Java tiene un ciclo para cada necesidad de programación.

while

El ciclo **while** es la sentencia de iteración más importante de Java. Con este ciclo se repite una sentencia o un bloque mientras la condición de control es verdadera. Su forma general es:

```
while (condición) {
  // cuerpo del ciclo
}
```

La *condición* puede ser cualquier expresión **booleana**. El cuerpo del ciclo se ejecutará mientras la expresión condicional sea verdadera. Cuando la *condición* sea falsa, la ejecución pasa a la siguiente línea de código localizada inmediatamente después del ciclo. Las llaves no son necesarias si solamente se repite una sentencia en el cuerpo del ciclo.

El ciclo **while** que se presenta a continuación cuenta hacia atrás comenzando en 10 e imprime exactamente diez líneas con la palabra"tick":

```
// Ejemplo de un ciclo while.
class While {
  public static void main (String args[]) {
    int n = 10;

    while (n > 0) {
      System.out.println ("tick " + n);
      n--;
    }
  }
}
```

Cuando se ejecuta este programa, la salida es:

```
tick 10
tick 9
tick 8
tick 7
tick 6
tick 5
tick 4
tick 3
tick 2
tick 1
```

Ya que el ciclo **while** evalúa la expresión condicional al inicio del ciclo, el cuerpo del mismo no se ejecutará nunca si al comenzar la condición es falsa. Por ejemplo, en el siguiente fragmento, la llamada a **println ( )** no se ejecuta nunca:

```
int a = 10, b = 20;

while (a > b)
  System.out.println ("Esto no se mostrará");
```

El cuerpo del ciclo **while** (o de cualquier otro ciclo de Java) puede estar vacío ya que una *sentencia nula*, que consiste únicamente en un punto y coma, es sintácticamente válida en Java. Considere, por ejemplo, las siguientes líneas de código:

```
// El cuerpo de un ciclo puede estar vacío.
class SinCuerpo {
  public static void main (String args[]) {
    int i, j;

    i = 100;
    j = 200;

    // Para localizar el punto medio entre i y j
    while (++i < --j); // no existe el cuerpo en este ciclo
    System.out.println ("El punto medio es " + i);
  }
}
```

Este programa encuentra el punto medio entre **i** y **j**. La salida que se genera es la siguiente:

```
El punto medio es 150
```

El ciclo **while** funciona de la siguiente manera: el valor de **i** se incrementa y el valor de **j** se reduce. A continuación se comparan estos valores. Si el nuevo valor de **i** es aún menor que el nuevo valor de **j**, entonces el ciclo se repite. Si **i** es igual o mayor que **j**, el ciclo se detiene. Al salir del ciclo, **i** mantendrá un valor intermedio entre los valores iniciales de **i** y **j**. (Naturalmente este procedimiento sólo funciona cuando al comenzar **i** es menor que **j**). Como ha visto, no es necesario que exista un cuerpo del ciclo; en este caso todas las acciones se producen dentro de la propia expresión condicional. En programas profesionales escritos en Java es frecuente encontrar ciclos cortos sin ningún cuerpo cuando se pueden introducir en la expresión lógica que controla el ciclo todas las acciones necesarias.

do-while

Como se acaba de ver, si la expresión condicional que controla un ciclo **while** es inicialmente falsa, el cuerpo del ciclo no se ejecutará ni una sola vez. Sin embargo, puede haber casos en los que se quiera ejecutar el cuerpo del ciclo al menos una vez, incluso cuando la expresión condicional sea inicialmente falsa. En otras palabras, puede que se desee evaluar la expresión condicional al final del ciclo, en lugar de hacerlo al principio. Afortunadamente, Java dispone de un ciclo que lo hace exactamente así, el ciclo **do-while**. El ciclo **do-while** ejecuta siempre, al menos una vez, el cuerpo, ya que la expresión condicional se encuentra al final. Su forma general es:

```
do {
  // cuerpo del ciclo
} while (condición);
```

En cada iteración del ciclo **do-while** se ejecuta en primer lugar el cuerpo del ciclo, y a continuación se evalúa la expresión condicional. Si la expresión es verdadera, el ciclo se repetirá. En caso contrario, el ciclo finalizará. Como en todos los demás ciclos de Java, la *condición* debe ser una expresión **booleana**.

Las siguientes líneas son un ejemplo de un ciclo **do-while**. El ejemplo es otra versión del programa "tick" y genera la misma salida que se obtuvo anteriormente.

```
// Ejemplo del ciclo do-while.
class DoWhile {
  public static void main (String args[]) {
    int n = 10;

    do {
      System.out.println ("tick " + n);
      n--;
    } while (n > 0);
  }
}
```

Aunque el ciclo que se acaba de presentar en las líneas anteriores es técnicamente correcto, se puede escribir de una manera más eficiente:

```
do {
  System.out.println ("tick " + n);
} while (--n > 0);
```

En este ejemplo, en la expresión **(--n > 0)** se combina el decremento de **n** y la comparación de la misma variable **n** con cero en una única expresión. Esto se realiza de la siguiente forma: en primer lugar, la sentencia --**n** reduce el valor de **n** y devuelve el nuevo valor de **n**; este valor se compara con cero, si es mayor que cero el ciclo continúa, y en caso contrario, finaliza.

El ciclo **do-while** es muy útil cuando se procesa un menú de selección, ya que normalmente se desea que el cuerpo del menú se ejecute al menos una vez. Considere el siguiente programa en el que se implementa un sistema de ayuda muy sencillo para las sentencias de selección e iteración de Java:

```
// Uso de un ciclo do-while para procesar un menú de selección
c1ass Menu {
  public static void main (String args[])
    throws java.io.IOException {
    char eleccion;

    do {
      System.out.println ("Ayuda para:");
      System.out.println (" 1. if");
      System.out.println (" 2. switch");
      System.out.println (" 3. while");
      System.out.println (" 4. do-while");
      System.out.println (" 5. for\n");
      System.out.println ("Elige una opción:");
      eleccion = (char) System.in.read();
    } while (eleccion < '1' || eleccion > '5');

    System.out.println ("\n");

    switch (eleccion) {
      case '1':
        System.out.println ("La sentencia if:\n");
        System.out.println ("if (condición) sentencia;");
        System.out.println ("else sentencia;");
        break;
      case '2':
        System.out.println ("La sentencia switch:\n");
        System.out.println ("switch (expresion) {");
        System.out.println (" case constante:");
        System.out.println (" conjunto de sentencias");
        System.out.println (" break;");
        System.out.println (" // ...");
        System.out.println ("}");
        break;
      case '3':
        System.out.println ("La sentencia while:\n");
        System.out.println ("while (condición) sentencia;");
        break;
      case '4':
        System.out.println ("La sentencia do-while:\n");
        System.out.println ("do {");
        System.out.println ("sentencia;");
        System.out.println ("} while (condición);");
        break;
```

```
      case '5':
        System.out.println ("La sentencia for:\n");
        System.out.print ("for (inicialización; condición; iteración)");
        System.out.println (" sentencia;");
        break;
    }
  }
}
```

La salida que se genera con este programa es la siguiente:

```
Ayuda para:
  1. if
  2. switch
  3. while
  4. do-while
  5. for
Elige una opción:
4
La sentencia do-while:
do {
  sentencia;
} while (condición);
```

En este programa el ciclo **do-while** se utiliza para verificar que el usuario ha elegido una opción válida. En caso contrario, se vuelven a presentar al usuario todas las opciones. Ya que el menú se debe presentar al menos una vez, el ciclo **do-while** es el más indicado para llevar esto a cabo.

Otros elementos interesantes en este ejemplo son los siguientes: observe que los caracteres se leen desde el teclado mediante la instrucción **System.in.read ( ).** Ésta es una de las funciones de Java que permiten introducir datos desde el teclado. Aunque los métodos de E/S de datos por consola de Java no serán discutidos en detalle sino hasta el Capítulo 13, **System.in.read ( )** se utiliza aquí para obtener la elección del usuario. Esta función permite leer caracteres desde una entrada estándar (estos caracteres se devuelven como enteros lo que permite asignarlos a la variable **char**). Por omisión, la entrada estándar tiene un buffer, y esto obliga a presionar la tecla ENTER antes de que cualquier carácter escrito sea enviado al programa.

La entrada de datos por consola en Java es bastante limitada e incómoda. Además, la mayor parte de los programas y applets profesionales en Java son gráficos y basados en el sistema de ventanas. Por estas razones, en este libro no se ha hecho mucho uso de la entrada por consola. Sin embargo, es útil en este contexto. Otro punto de interés es el siguiente: Como se está utilizando la función **System.in.read( )**, el programa debe especificar la cláusula **throws java.io.IOException**. Esta línea es necesaria para la gestión de los errores que se produzcan en la entrada de datos. Esto es parte de las características que tiene Java para la gestión de excepciones, las cuales serán analizadas en el Capítulo 10.

for

En el Capítulo 2 se presentó un ejemplo sencillo del ciclo **for**. Como se podrá comprobar el ciclo **for** es una construcción versátil y potente.

Comenzando con JDK 5, existen dos formas del ciclo **for**. La primera forma es la tradicional que se ha utilizado desde la versión original de Java. La segunda es una forma nueva conocida

como"for-each". Ambos tipos de ciclos **for** son explicados a detalle aquí, comenzando con la forma tradicional.

La forma general de la sentencia **for** tradicional es la siguiente:

```
for (inicialización; condición; iteración) {
  // cuerpo
}
```

Si solamente se repite una sentencia, no es necesario el uso de las llaves.

El ciclo **for** actúa como se describe a continuación: cuando comienza, se ejecuta la parte de *inicialización*. Generalmente, la inicialización es una expresión que establece el valor de la *variable de control del ciclo,* que actúa como un contador que lo controla. Es importante comprender que la expresión de inicialización se ejecuta una sola vez. A continuación, se evalúa la *condición*, que debe ser una expresión **booleana** mediante la que, normalmente, se compara la variable de control con un valor de referencia. Si la expresión es verdadera, entonces se ejecuta el cuerpo del ciclo. Si es falsa, el ciclo finaliza. A continuación se ejecuta la parte correspondiente a la *iteración.* Habitualmente ésta es una expresión en la que se incrementa o reduce el valor de la variable de control. Cada vez que se recorre el ciclo, en primer lugar se vuelve a evaluar la expresión condicional, a continuación se ejecuta el cuerpo y después la expresión de iteración. Este proceso se repite hasta que la expresión condicional sea falsa.

A continuación otra versión del programa"tick", ahora utilizando un ciclo **for**:

```
// Ejemplo del ciclo for
class ForTick {
  public static void main (String args[]) {
    int n;

    for (n=10; n>0; n--)
      System.out.println ("tick " + n);
  }
}
```

Declaración de variables de control dentro del ciclo

A menudo, la variable que controla el ciclo **for** sólo se necesita en el ciclo, y no se utiliza en ninguna otra parte. En este caso, es posible declarar esta variable en la sección de inicialización del **for**. Por ejemplo, el programa anterior se puede reescribir de forma que la variable de control del ciclo **n** se declare como **int** dentro del **for**:

```
// Declaración de la variable de control del ciclo dentro del for
class ForTick {
  public static void main (String args[]) {

    // aquí se declara n dentro del ciclo for
    for (int n=10; n>0; n--)
      System.out.println ("tick " + n);
  }
}
```

Cuando se declara una variable dentro del ciclo **for**, hay un punto importante que se ha de tener en cuenta: la vida de esa variable finaliza cuando lo hace la sentencia **for**, (es decir, el alcance de la variable está limitado al ciclo **for**). Fuera del ciclo **for** la variable no existirá. Si la

variable de control del ciclo interviene en alguna otra parte del programa, no se puede declarar dentro del ciclo **for**.

Cuando la variable de control no se va a utilizar en ninguna otra parte del código, la mayoría de programadores la declaran dentro del ciclo **for**. El programa que aparece a continuación es un programa sencillo para comprobar si un número es primo o no. Observe que la variable de control, **i**, se declara dentro del ciclo **for**, ya que no se utiliza en ninguna otra parte.

```
// Prueba de números primos
class NumeroPrimo {
  public static void main (String args[]) {
    int num;
    boolean esPrimo = true;

    num = 14;
    for (int i=2; i <= num/i; i++) {
      if ({num % i) == 0) {
        esPrimo = false;
        break;
      }
    }
    if (esPrimo) System.out.println ("El número es primo");
    else System.out.println ("El número no es primo");
  }
}
```

Uso del separador coma

En ocasiones puede ser necesario incluir más de una sentencia en las secciones de inicialización e iteración del ciclo **for**. Considere, por ejemplo, el ciclo que aparece en el siguiente programa:

```
class Ejemplo {
  public static void main (String args[]) {
    int a, b;

    b = 4;
    for (a=l; a<b; a++) {
      System.out.println ("a = " + a);
      System.out.println ("b = " + b);
      b--;
    }
  }
}
```

En este caso el ciclo está controlado por la interacción de dos variables y sería útil incluir ambas variables en la propia sentencia **for** en lugar de operar dentro del cuerpo sobre la variable **b.** Afortunadamente, Java proporciona la posibilidad de tener dos o más variables de control del ciclo **for** y permite que haya varias sentencias en las partes de inicialización e iteración. Cada sentencia se separa de la siguiente mediante una coma.

El ciclo **for** anterior se codifica de manera más eficiente de la siguiente forma:

```
/ / Utilización de la coma.
class Coma {
```

```
  public static void main (String args[]) {
    int a, b;

    for (a=1, b=4; a<b; a++, b--) {
      System.out.println ("a = " + a);
      System.out.println ("b = " + b);
    }
  }
}
```

En este ejemplo la parte de inicialización establece los valores de ambas variables **a** y **b** sólo una vez. Mientras que las dos sentencias separadas por comas en la sección de iteración se ejecutan cada vez que se repite el ciclo. El programa genera la siguiente salida:

```
a = 1
b = 4
a = 2
b = 3
```

NOTA *Si está familiarizado con C/C++, sabe que en estos lenguajes la coma es un operador que se puede utilizar en cualquier expresión válida. Sin embargo, en Java no ocurre lo mismo. En Java la coma es un separador.*

Algunas variaciones de los ciclos for

El ciclo **for** admite variaciones que aumentan su potencia y aplicabilidad. La razón de que sea tan flexible es que sus tres partes, la inicialización, la prueba condicional y la iteración no tienen por qué ser utilizados con ese único objetivo. De hecho, las tres secciones del **for** se pueden utilizar con otros fines. Veamos algunos ejemplos.

Una de las variaciones más comunes se refiere a la expresión condicional. Específicamente esta expresión no tiene como único objetivo comparar la variable de control con un valor específico. La condición que controla el ciclo **for** puede ser cualquier expresión **booleana**. Considere, por ejemplo, el siguiente fragmento de código:

```
boolean done = false;

for (int i=1; !done; i++) {
  / / ...
  if (interrupted ()) done = true;
}
```

En este ejemplo el ciclo **for** se repite hasta que la variable **booleana done** se hace **verdadera**, aquí no se compara el valor de **i** con un valor fijo.

A continuación se presenta otra variación interesante del ciclo **for**. La expresión de inicialización, o la de iteración, o ambas pueden estar ausentes, tal y como ocurre en el siguiente programa:

```
// Algunas partes del ciclo for pueden estar vacías.
class ForVar {
  public static void main (String args[]) {
    int i;

    boolean done = false;
```

```
    i = 0;
    for ( ; !done; ) { .
      System.out.println ("i es " + i);
      if (i == 10) done = true;
      i++;
    }
  }
}
```

En este caso, las expresiones de inicialización y de iteración se han eliminado del ciclo **for**, por lo que las partes correspondientes están vacías. Aunque esto no tiene importancia en este sencillo ejemplo —se podría considerar como un estilo de programación bastante pobre—, puede haber ocasiones en las que seguir este modelo tenga sentido. Por ejemplo, cuando la condición inicial es una expresión compleja y se establece en cualquier otra parte del programa o cuando la variable de control cambia de forma no secuencial en función de acciones que tienen lugar en el cuerpo del ciclo, podría ser conveniente dejar estas partes del **for** vacías.

Otra variación del ciclo **for** es la siguiente: se puede crear intencionalmente un ciclo infinito (un ciclo que nunca termina) si se dejan vacías las tres partes del ciclo **for**. Por ejemplo:

```
for( ; ; ) {
  //…
}
```

Este ciclo se ejecutará indefinidamente, ya que no hay ninguna condición que controle su finalización. Aunque hay programas en los que es necesario un ciclo infinito, como en los procesadores de órdenes del sistema operativo, la mayor parte de los "ciclos infinitos" son sólo ciclos con requerimientos especiales de finalización. Como se verá más adelante, existe una manera de terminar un ciclo, incluso un ciclo infinito, sin utilizar la expresión condicional normal del ciclo.

La versión for-each del ciclo for

Comenzando con JDK 5, una segunda forma del ciclo **for** fue definida para implementar un ciclo estilo "for-each". Como posiblemente sepa, la teoría contemporánea del lenguaje ha incluido el concepto "for-each" el cual rápidamente se ha convertido en una característica estándar que los programadores esperan esté presente en los lenguajes de programación. El ciclo estilo **for-each** está diseñado para iterar a través de una colección de objetos, como un arreglo por ejemplo, en estricto orden secuencial de inicio a final. A diferencia de algunos lenguajes como C#, que utilizan la palabra clave **foreach** para implementar este tipo de ciclos, Java agrega la capacidad **for-each** como parte de la sentencia **for**. La ventaja de este enfoque es que no se requieren nuevas palabras clave, y no demerita la funcionalidad del código preexistente. El estilo **for-each** del **for** es también llamado ciclo **for** *ampliado*.

La forma general del ciclo for-each se muestra a continuación:

for *(tipo variable : colección) bloque*

Donde *tipo* especifica el tipo y *variable* especifica el nombre de una *variable de iteración* que recibirá elementos de una colección, uno a la vez de principio a fin. La colección de elementos que serán recorridos se especifica por **colección**. Existen varios tipos de colecciones que pueden

ser usadas por el ciclo **for**, pero el único tipo que se usará en este capítulo es el arreglo (otros tipos de colecciones que pueden ser usadas con el ciclo **for** serán examinadas posteriormente en este libro).

Con cada iteración del ciclo, el siguiente elemento en la colección es recuperado y guardado en la *variable*. El ciclo se repite hasta que todos los elementos en la colección han sido recuperados.

Dado que la variable de iteración recibe valores de la colección, el *tipo* debe ser el mismo tipo que los elementos guardados en la colección o compatible con ellos. De forma que cuando se esté iterando sobre arreglos el *tipo* debe ser compatible con el **tipo** base del arreglo.

Para entender la motivación detrás del ciclo estilo for-each, veamos como quedaría un ciclo **for** tradicional equivalente. El siguiente fragmento de código utiliza un ciclo **for** tradicional para calcular la suma de los valores en un arreglo:

```
int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
int sum = 0

for (int i=0; i < 10; i++) sum += nums[i];
```

Para calcular la suma, cada elemento en **nums** es leído de forma secuencial del inicio al final. Esto se logra por la indexación manual del arreglo **nums**, mediante la variable de control de ciclo **i**.

El estilo for-each automatiza el ciclo **for** anterior. Específicamente, elimina la necesidad de establecer un contador de ciclo, declarar un valor de inicio y uno de fin, y manualmente indexar el arreglo. En lugar de esto, automáticamente el ciclo recorre el arreglo completo, obteniendo un elemento a la vez en secuencia de principio al fin. Por ejemplo, a continuación se muestra el programa anterior utilizando el estilo del ciclo for-each:

```
int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
int sum = 0

for (int x: nums) sum += x;
```

Con cada iteración, a **x** se da un valor igual al siguiente elemento en **nums**. De forma que en la primera iteración, **x** contiene 1, en la segunda iteración contiene 2, y así sucesivamente. La sintaxis no sólo es eficiente, además previene y limita errores.

El siguiente programa es un ejemplo completo que demuestra el uso del ciclo estilo for-each:

```
// Ejemplo del ciclo estilo for-each.
class ForEach {
  public static void main(String args[]) {
    int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    int sum = 0;

    // Uso del ciclo estilo for-each para mostrar y sumar valores
    for(int x : nums) {
      System.out.println("El valor es: " + x);
      sum += x;
    }

    System.out.println("La suma de los valores es: " + sum);
  }
}
```

La salida del programa es la siguiente:

```
El valor es: 1
El valor es: 2
El valor es: 3
El valor es: 4
El valor es: 5
El valor es: 6
El valor es: 7
El valor es: 8
El valor es: 9
El valor es: 10
La suma de los valores es: 55
```

Como lo muestra esta salida, el ciclo estilo for-each automáticamente pasa a través del arreglo en secuencia desde el elemento con índice inferior hasta el elemento con índice superior.

Aunque el ciclo estilo for-each itera hasta que todos los elementos del arreglo han sido examinados, es posible terminar el ciclo antes utilizando una sentencia **break**. Por ejemplo en este programa sólo se suman los primeros 5 elementos del arreglo **nums**:

```
// Utilizando break dentro de un ciclo estilo for-each
class ForEach2 {
  public static void main(String args[]) {
    int sum = 0;
    int nums[] = { 1,2,3,4,5,6,7,8,9,10 };

    // Se utiliza el ciclo estilo for-each para mostrar y sumar los valores
    for(int x : nums) {
      System.out.println ("El valor es: " + x);
      sum += x;
      if(x == 5) break; // detener el ciclo en el quinto elemento del arreglo
    }
    System.out.println("La suma de los cinco primeros valores es: " + sum);
  }
}
```

La salida del programa es:

```
El valor es: 1
El valor es: 2
El valor es: 3
El valor es: 4
El valor es: 5
La suma de los cinco primeros valores es: 15
```

Como es evidente, el ciclo **for** termina después de que el quinto elemento ha sido obtenido del arreglo. La sentencia **break** también puede utilizarse con otros ciclos de Java, pero esto será tratado a detalle más adelante en este capítulo.

Existe un punto importante a entender acerca del ciclo estilo for-each. La variable de iteración es de "sólo lectura" dado que se relaciona directamente con los valores del arreglo. Una asignación a la variable de iteración no tiene efecto en los valores del arreglo. En otras palabras,

no es posible cambiar el contenido del arreglo asignado a la variable de iteración un nuevo valor. Por ejemplo, considere este programa:

```
// La variable de iteración el ciclo for-each es esencialmente de sólo lectura.
class SinCambios {
  public static void main(String args[]) {
    int nums [] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

    for(int x : nums) {
      System.out.print(x + " ");
      x = x * 10; // no tiene efecto en la suma
    }

    System.out.println() ;

    for(int x : nums)
      System.out.print(x + " ");

    System.out.println() ;

  }
}
```

El primer ciclo **for** incrementa el valor de la variable de iteración en un factor de 10. Sin embargo, dicha asignación no tiene efecto en el contenido del arreglo **nums**, como lo muestra el segundo ciclo **for**. La salida mostrada a continuación prueba la afirmación anterior:

```
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
```

Iterando sobre arreglos multidimensionales

La versión ampliada del ciclo **for** también funciona con arreglos multidimensionales. Sin embargo, recordemos que en Java los arreglos multidimensionales consisten en *arreglos de arreglos*. Por ejemplo, un arreglo de dos dimensiones es un arreglo de arreglos unidimensionales. Esto es importante, cuando se itera sobre arreglos multidimensionales, porque cada iteración obtiene el siguiente arreglo, no un elemento individual. Además, la variable de iteración en el ciclo **for** debe ser compatible con el tipo del arreglo del que se están obteniendo elementos. Por ejemplo, en el caso de un arreglo de dos dimensiones, la variable de iteración debe ser una referencia a un arreglo unidimensional. En general, cuando se usa el ciclo for-each para iterar sobre un arreglo de *N* dimensiones, los objetos obtenidos serán arreglos de *N-1* dimensiones. Para comprender las implicaciones de esto, considere el siguiente programa, el cual utiliza ciclos **for** anidados para obtener los elementos de un arreglo bidimensional renglón por renglón del primero al último.

```
// Utilizando un ciclo estilo for-each con un arreglo bidimensional
class ForEach3 {
  public static void main(String args[]) {
    int sum = 0;
    int nums [] [] = new int [3] [5] ;

    // Introduce algunos valores en el arreglo nums
    for(int i = 0; i < 3; i++)
```

```
    for(int j=0; j < 5; j++)
      nums[i] [j] = (i+1)*(j+1);

  // uso del ciclo estilo for-each para mostrar en pantalla
  la suma de los valores
    for(int x[] : nums) {
      for(int y : x) {
        System.out.println("El valor es: " + y);
        sum += y;
      }
    }
    System.out.println("El suma de los valores es: " + y);
  }
}
```

La salida del programa se muestra a continuación:

```
El valor es: 1
El valor es: 2
El valor es: 3
El valor es: 4
El valor es: 5
El valor es: 2
El valor es: 4
El valor es: 6
El valor es: 8
El valor es: 10
El valor es: 3
El valor es: 6
El valor es: 9
El valor es: 12
El valor es: 15
La suma de los valores es: 90
```

Es importante poner especial atención en la línea siguiente en el programa:

```
for (int x[] : nums) {
```

Observe cómo está declarada. Ésta es una referencia a un arreglo unidimensional de enteros. Esto es necesario debido a que cada iteración del ciclo **for** obtiene el siguiente *arreglo* en **nums**, comenzando con el arreglo especificado como **nums[0]**. El ciclo **for** interior itera a través de cada uno de esos arreglos, mostrando los valores de sus elementos.

Aplicando el ciclo for ampliado

Dado que el ciclo estilo for-each solamente puede recorrer un arreglo secuencialmente de principio a fin, es posible pensar que su uso es limitado, lo cuál no es cierto. Una amplia gama de algoritmos requiere exactamente este mecanismo. Entre los más comunes se encuentran los algoritmos de búsqueda. Por ejemplo, el siguiente programa utiliza un ciclo **for** para buscar un valor dentro de un arreglo no ordenado; el ciclo se detiene cuando el valor es encontrado.

```
// Ejemplo de búsqueda en un arreglo utilizando un ciclo estilo for-each
class Search {
  public static void main(String args[]) {
    int nums[] = { 6, 8, 3, 7, 5, 6, 1, 4 };
    int val = 5;
    boolean found = false;

    // Utiliza el ciclo estilo for-each para buscar val en el arreglo nums.
    for(int x : nums) {
      if (x == val) {
        found = true;
        break;
      }
    }

    if (found)
      System.out.println("Valor encontrado");
  }
}
```

El ciclo estilo for-each es una excelente opción en esta aplicación porque la búsqueda en un arreglo no ordenado involucra examinar cada elemento en secuencia. Claro está que si el arreglo estuviera ordenado, una búsqueda binaria podría ser utilizada, lo cual requeriría otro tipo de ciclo. Otro tipo de aplicaciones que se benefician del ciclo estilo for-each incluye el cálculo de un promedio, encontrar el mínimo o el máximo de un conjunto, la búsqueda de duplicados, y muchas más.

Aunque hemos estado usando arreglos en los ejemplos de este capítulo, el ciclo estilo for-each es especialmente útil cuando se trabaja con colecciones; las colecciones serán descritas en la Parte II de este libro. Generalmente el ciclo **for** puede iterar a través de los elementos de cualquier colección de objetos, mientras que la colección satisfaga ciertas restricciones, las cuales serán descritas en el capítulo 17.

Ciclos anidados

Como en otros lenguajes de programación, Java permite el anidamiento de ciclos. Es decir, un ciclo puede estar dentro de otro. Las siguientes líneas son un ejemplo de ciclos **for** anidados:

```
// Ejemplo de ciclos anidados.
class Anidados {
  public static void main (String args[]) {
    int i. j;

    for(i=0; i<l0; i++) {
      for(j=i; j<l0; j++)
        System.out.print (".");
      System.out.println ();
    }
  }
}
```

La salida que produce este programa se muestra a continuación:

```
..........
.........
........
.......
......
.....
....
...
..
.
```

Sentencias de salto

Java incorpora tres sentencias de salto: **break, continue** y **return**. Estas sentencias transfieren el control a otra parte del programa. Cada una es examinada aquí.

NOTA *Además de las sentencias de salto que se analizan en este punto, el lenguaje Java permite alterar el flujo del programa de otra manera: a través de la gestión de excepciones. La gestión de excepciones proporciona un método estructurado por el cual los errores en tiempo de ejecución son capturados y gestionados por el programa. Las palabras clave en las que se apoya la gestión de excepciones son* ***try***, ***catch***, ***throw***, ***throws*** *y* ***finally****. En esencia, el mecanismo de gestión de excepciones, permite al programa realizar una bifurcación externa. Como la gestión de excepciones es un concepto muy extenso, será discutido en su propio capítulo, el Capítulo 10.*

break

La sentencia **break** tiene tres usos en Java. En primer lugar, tal y como se ha visto, finaliza una secuencia de sentencias dentro de una sentencia **switch**. En segundo lugar se puede utilizar para salir de un ciclo.Y en tercer lugar se puede usar como una forma"civilizada"de la instrucción goto. A continuación se presentan ejemplos de los dos últimos usos.

Uso de la sentencia break para salir de un ciclo

Mediante la sentencia **break** se puede forzar la finalización inmediata de un ciclo, evitando la expresión condicional y el resto de código dentro del cuerpo del ciclo. Cuando se encuentra una sentencia **break** dentro de un ciclo, el ciclo termina y el control del programa se transfiere a la sentencia que sigue al ciclo. Por ejemplo:

```
// Uso de break para salir de un ciclo.
class BreakLoop {
  public static void main (String args[]) {
    for (int i=0; i<100; i++) {
      if (i == 10) break; // el ciclo finaliza si i es igual a 10
      System.out.println ("i: " + i);
    }
    System.out.println ("Ciclo completado.");
  }
}
```

Este programa genera la siguiente salida:

```
i: 0
i: 1
i: 2
i: 3
i: 4
i: 5
i: 6
i: 7
i: 8
i: 9
Ciclo completado.
```

Aunque el ciclo **for** se ha diseñado para ejecutarlo desde los valores de i =0 a i =99, la sentencia **break** hace que finalice antes, cuando **i** es igual a 10.

La sentencia **break** puede utilizarse con cualquier ciclo del lenguaje Java, incluyendo ciclos diseñados intencionalmente como ciclos infinitos. A continuación se presenta el programa anterior utilizando esta vez un ciclo **while**. La salida de este programa es la misma que la que se acaba de mostrar.

```
// Uso de break para salir de un ciclo while.
class BreakLoop2 {
  public static void main (String args[]) {
    int i =0;

    while (i < 100) {
      if (i == 10) break; // El ciclo se termina si i es igual a 10
      System.out.println ("i: " + i);
      i++;
    }
    System.out.println ("Ciclo completado.");
  }
}
```

Cuando la sentencia **break** se utiliza dentro de un conjunto de ciclos anidados, solamente se saldrá del ciclo más interno. Por ejemplo:

```
// Uso del break con ciclos anidados.
class BreakLoop3 {
  public static void main (String args[]) {
    for (int i=0; i<3; i++) {
      System.out.print ("Paso " + i + ": ");
      for (int j=0; j<l00; j++) {
        if (j == 10) break; // el ciclo finaliza si i es igual a 10
        System.out.print (j + " ");
      }
      System.out.println ();
    }
    System.out.println ("Ciclo completado.");
  }
}
```

Este programa genera la siguiente salida:

```
Paso 0: 0 1 2 3 4 5 6 7 8 9
Paso 1: 0 1 2 3 4 5 6 7 8 9
Paso 2: 0 1 2 3 4 5 6 7 8 9
Ciclo completado.
```

Como se puede ver, la sentencia **break** en el ciclo interior sólo causa la finalización de ese ciclo. El ciclo exterior no resulta afectado.

Hay que tener en cuenta otros dos puntos importantes sobre la sentencia **break**. En primer lugar, que dentro de un ciclo puede haber más de una sentencia **break**, y conviene ser cuidadoso ya que muchas sentencias **break** dentro de un programa tienden a quitarle estructura al código. Y en segundo lugar, la sentencia **break** que finaliza una sentencia **switch** afecta sólo a esa sentencia switch y no a los ciclos que pudieran estar conteniendo a la sentencia **switch**.

PARA RECORDAR *La sentencia break no fue diseñada con el propósito de ser la finalización normal de un ciclo. La expresión condicional del ciclo es la que tiene ese objetivo. La sentencia break debe utilizarse para cancelar un ciclo sólo cuando se produce algún tipo de situación especial.*

Uso de la sentencia break como una forma de goto

Además de su uso con la sentencia **switch** y los ciclos, la sentencia **break** también se puede utilizar como una forma "civilizada" de la sentencia goto. Java no incorpora a la instrucción **goto** en el lenguaje ya que su uso da lugar a una forma de realizar bifurcaciones arbitrarias y no estructuradas. Un código que incorpora la sentencia goto es difícil de entender y mantener. También impide al compilador realizar ciertas optimizaciones. Existen, sin embargo, algunas ocasiones en las que la sentencia goto es una construcción válida y legítima para establecer el control del flujo. Por ejemplo, el goto puede ser útil cuando se trata de salir de un conjunto de ciclos profundamente anidados. Para gestionar tales situaciones, Java define una forma expandida de la sentencia **break**, la cual permite salir de uno o más bloques de código. Y no es necesario que estos bloques sean parte de un ciclo o una sentencia **switch**. Pueden ser cualquier tipo de bloque. Además, se puede especificar de forma precisa en qué parte de programa se reanudará la ejecución, ya que esta forma de la sentencia break actúa con una etiqueta. Esta forma de la sentencia **break** tiene las ventajas de un goto pero sin sus inconvenientes.

La forma general de una sentencia **break** con etiqueta es la siguiente:

break *etiqueta;*

Donde *etiqueta* es el nombre de una etiqueta que identifica un bloque de código. Cuando se ejecuta esta forma de la sentencia break, el control se transfiere fuera del bloque etiquetado que debe encerrar la sentencia **break**. No es necesario que este bloque sea el que encierra inmediatamente al break. Esto significa que se puede utilizar una sentencia **break** etiquetada para salir de un conjunto de ciclos anidados. Sin embargo, no se puede transferir el control a un bloque del código que esté encerrando a la sentencia **break**.

Para etiquetar un bloque se pone una etiqueta en el comienzo del mismo. Una *etiqueta* es cualquier identificador válido de Java seguido por dos puntos. Una vez que el bloque está etiquetado, se puede utilizar la etiqueta como término o destino de una sentencia **break**. La ejecución se reanuda al *final* del bloque etiquetado. El siguiente programa muestra tres bloques anidados, cada uno con su propia etiqueta. La sentencia **break** hace que la ejecución se adelante y continúe inmediatamente después del bloque etiquetado como **segundo**, saltando las dos sentencias **println( ).**

```
// Uso de la sentencia break como forma civilizada de goto.
class Break {
  public static void main (String args[]) {
    boolean t = true;

    primero: {
      segundo: {
        tercero: {
          System.out.println ("Antes del break.");
          if (t) break segundo; // sale fuera del bloque "segundo"
          System.out.println ("Esto no se ejecutará");
        }
        System.out.println ("Esto no se ejecutará");
      }
      System.out.println("Esto va después del segundo bloque.");
    }
  }
}
```

Al ejecutarse este programa se genera la siguiente salida:

```
Antes del break.
Esto va después del segundo bloque.
```

Uno de los usos más frecuentes de la sentencia **break** con etiqueta es salir de ciclos anidados. Por ejemplo, en el siguiente programa, el ciclo exterior se ejecuta una sola vez:

```
// Uso de la sentencia break para salir de ciclos anidados.
class BreakLoop4 {
  public static void main (String args[]) {
    exterior: for(int i=0; i<3; i++) {
      System.out.print ("Paso " + i + ": ");
      for (int j=0; j<100; j++) {
        if (j == 10) break exterior; // sale de ambos ciclos
        System.out.print (j + " ");
      }
      System.out.println ("Esto no se imprimirá");
    }
    System.out.println ("Ciclos completados.");
  }
}
```

Este programa genera la siguiente salida:

```
    Paso 0: 0 1 2 3 4 5 6 7 8 9 Ciclos completados.
```

Como se puede ver, cuando en el ciclo interior se llega a la sentencia break, ambos ciclos terminan. Note que en este ejemplo se etiqueta a la sentencia **for** con su bloque de código incluido.

Se debe tener en cuenta que no es posible usar la sentencia break a cualquier etiqueta. Sólo es posible utilizar la sentencia break con etiqueta cuando la etiqueta está definida dentro del bloque código. El siguiente programa, por ejemplo, no es válido y no compilará:

```
// Este programa contiene un error.
class BreakError {
```

```
  public static void main(String args[]) {
    uno: for (int i=0; i<3; i++) {
      System.out.print ("Paso " + i + ":" );
    }
    for (int j=0; j<l00; j++) {
      If (j == 10) break uno; // ERROR
        System.out.print (j + " ");
    }
  }
}
```

Como el lazo etiquetado con **uno** no encierra la sentencia **break**, no se puede transferir el control a ese bloque.

continue

Algunas veces es útil forzar una nueva iteración de un ciclo. Esto es, continuar ejecutando el ciclo pero sin concluir completamente el procesamiento de la iteración actual, o dicho de otra forma, que se produzca un salto desde el cuerpo del ciclo al final del ciclo. La sentencia que permite tal acción es la sentencia **continue**. En los ciclos **while** y **do-while**, una sentencia **continue** hace que el control se transfiera directamente a la expresión condicional que controla el ciclo. En un ciclo **for**, va en primer lugar a la parte de la iteración de la sentencia **for** y después a la expresión condicional. En cualquiera de los tres ciclos, se ignora cualquier código intermedio.

El siguiente ejemplo utiliza la sentencia **continue** para hacer que se impriman dos números en cada línea:

```
// Ejemplo de la sentencia continue.
class Continue {
  public static void main (String args[]) {
    for(int i=0; i<l0; i++) {
      System.out.print (i + " ");
      if (i%2 == 0) continue;
      System.out.println ("");
    }
  }
}
```

Este código utiliza el operador % para comprobar si **i** es par. Si es así, el ciclo continúa sin imprimir una nueva línea. La salida de este programa es la siguiente:

```
0 1
2 3
4 5
6 7
8 9
```

La sentencia **continue**, al igual que la sentencia **break**, puede definir una etiqueta para indicar qué ciclo es el que debe **continuar**. En el siguiente ejemplo se utiliza la sentencia **continue** para imprimir una tabla de multiplicación triangular del 0 al 9.

```
// Utilización de la sentencia continue con una etiqueta.
class ContinueLabel {
  public static void main(String args[]) {
exterior: for (int i=0; i<10; i++) {
           for (int j=0; j<10; j++) {
             if (j > i) {
               System.out.println ();
               continue exterior;
             }
           System.out.print (" " + (i * j));
         }
       }
       System.out.println ();
  }
}
```

La sentencia **continue** en este ejemplo finaliza el ciclo que tiene como variable de control **j** y continúa con la siguiente iteración del ciclo que tiene como variable de control **i**. La salida de este programa es la siguiente:

```
0
0 1
0 2 4
0 3 6 9
0 4 8 12 16
0 5 10 15 20 25
0 6 12 18 24 30 36
0 7 14 21 28 35 42 49
0 8 16 24 32 40 48 56 64
0 9 18 27 36 45 54 63 72 81
```

Es raro encontrar un uso adecuado para la sentencia **continue**. Una razón es que Java proporciona un conjunto de sentencias de ciclo que se ajustan a la mayor parte de aplicaciones. Sin embargo, en aquellas circunstancias especiales en las que sea necesaria una nueva iteración, la sentencia **continue** permite hacerlo de una forma estructurada.

return

La última sentencia de control es **return**. Se utiliza para salir explícitamente de un método, es decir, hace que el control del programa vuelva al método llamante. Por este motivo, esta sentencia se clasifica como una sentencia de salto. Aunque esta sentencia se discutirá con mayor detalle en el Capítulo 6, a continuación se presenta un ejemplo sencillo de introducción.

En cualquier momento, en un método, la sentencia **return** se puede utilizar para hacer que la ejecución regrese al método llamante. Además, la sentencia **return** hace que finalice inmediatamente el método en el que se ejecuta. Este punto se ilustra con el siguiente ejemplo, donde la sentencia **return** hace que la ejecución vuelva al intérprete Java, que es quién llama a **main ().**

```
// Ejemplo de la sentencia return.
class Return {
  public static void main (String args[]) {
    boolean t = true;
```

```
    System.out.println ("Antes de return.");

    If (t) return; // vuelve al método llamante

    System.out.println ("Esto no se ejecutará.");
  }
}
```

La salida del programa es la siguiente:

```
    Antes de return.
```

Como puede verse la última sentencia **println( )** no se ejecuta. Cuando se ejecuta el **return**, el control vuelve al método llamante.

Un último punto a tener en cuenta en el programa anterior es que la sentencia **if(t)** es necesaria. Sin ella, el compilador Java generará un error al saber que la última sentencia **println( )** nunca se ejecutaría. Para evitar que se produzca este error, mediante la sentencia **if** engañamos al compilador.

CAPÍTULO 6

Clases

Las clases son el núcleo de Java. Es la construcción lógica sobre la que se basa el lenguaje Java porque define la forma y naturaleza de un objeto. De tal forma que son la base de la programación orientada a objetos en Java. Cualquier concepto que se quiera implementar en Java debe estar encapsulado dentro de una clase.

Dada la importancia que tienen las clases en Java, este capítulo y los próximos se dedican a este tema. Aquí introduciremos los elementos básicos de una clase y aprenderemos cómo se usan las clases para crear objetos. También veremos los métodos, constructores y la palabra clave **this**.

Fundamentos de clases

Las clases se han utilizado desde el comienzo de este libro. Sin embargo, hasta ahora se habían utilizado sólo de una forma muy rudimentaria. Las clases creadas en los capítulos anteriores existían simplemente para encapsular el método **main( )**, que ha permitido mostrar los fundamentos de la sintaxis de Java. Como veremos, las clases son sustancialmente más potentes que las presentadas hasta el momento.

Probablemente la característica más importante de una clase es que define un nuevo tipo de dato. Una vez definido, este nuevo tipo de dato se puede utilizar para crear objetos de ese tipo o clase. De este modo, una clase es un *template* (un modelo) para un objeto, y un objeto es una *instancia* de una clase. Debido a que un objeto es una instancia de una clase, a menudo las dos palabras *objeto* e *instancia* se usan indistintamente.

La forma general de una clase

Cuando se define una clase, se declara su forma y naturaleza exactas, especificando los datos que contiene y el código que opera sobre esos datos. Las clases más sencillas pueden contener solamente código o solamente datos, pero, en la práctica, la mayoría de las clases contienen datos y código. Como veremos, el código de una clase define la interfaz con sus datos.

Una clase se declara mediante la palabra clave **class**. Las clases que se han utilizado hasta el momento son realmente ejemplos muy limitados de su forma completa. Las clases pueden ser (y normalmente lo son), mucho más complejas. La forma general de definir una **clase** es la siguiente:

```
class nombre_de_clase {
   tipo variable_de:instancia1;
   tipo variable_de_instancia2;
```

```
    // ...
    tipo variable_de_instanciaN;

    tipo nombre_de_método1 (parámetros) {
      // cuerpo del método
    }
    tipo nombre_de_método2 (parámetros) {
    // cuerpo del método
  }
  // ...
  tipo nombre_de_metodoN (parámetros) {
    // cuerpo del método
  }
}
```

Los datos, o variables, definidos en una **clase** se denominan *variables de instancia*. El código está contenido en los *métodos*. El conjunto de los métodos y las variables definidos dentro de una clase se denominan *miembros* de la clase. En la mayor parte de las clases, los métodos definidos acceden y actúan sobre las variables de instancia, es decir, los métodos determinan cómo se deben utilizar los datos de una clase.

Las variables definidas en una clase se llaman variables de instancia porque cada instancia de la clase (esto es, cada objeto de la clase), contiene su propia copia de estas variables. Así, los datos de un objeto son distintos y únicos de los de otros. Éste es un concepto importante sobre el que volveremos más adelante.

Todos los métodos tienen el mismo formato general, similar al del método **main( )** que hemos estado utilizando hasta el momento. Sin embargo, la mayor parte de los métodos no se especifican como **static** o **public**. Observe que la forma general de una clase no especifica un método **main( ).** Las clases de Java no tienen necesariamente un método **main( )**. Solamente se requiere un método **main( )** si esa clase es el punto de inicio del programa. Los applets no requieren un método **main( )**.

NOTA *Si usted está familiarizado con C++, observará que en Java, la declaración de una clase y la implementación de los métodos se almacenan en el mismo sitio y no se definen separadamente. Esto, en ocasiones, da lugar a archivos* ***.java*** *muy largos, ya que cualquier clase debe estar definida completamente en un solo archivo. Esta característica de diseño se estableció en Java, ya que se supuso que, a largo plazo, tener en un sólo sitio las especificaciones, declaraciones e implementación daría como resultado un código más fácil de mantener.*

Una clase simple

Comencemos nuestro estudio con un ejemplo sencillo, la clase denominada **Caja**. Esta clase define tres variables de instancia: **ancho**, **alto** y **largo**. En este caso, **Caja** no contiene método alguno, más adelante los añadiremos.

```
c1ass Caja {
  double ancho;
  double alto;
```

```
  double largo;
}
```

Como se ha dicho anteriormente, una clase define un nuevo tipo de dato. En este caso, el nuevo tipo se llama **Caja**. Utilizaremos este nombre para declarar objetos de tipo **Caja**. Es importante recordar que la declaración de una **clase** solamente crea un modelo o patrón y no un objeto real. Así que el código anterior no crea ningún objeto de la clase **Caja**.

Para crear un objeto de tipo **Caja** habrá que utilizar una sentencia como la siguiente:

```
Caja miCaja = new Caja(); // crea un objeto de la clase Caja llamado miCaja
```

Cuando se ejecute esta sentencia, **miCaja** será una referencia a una instancia de **Caja**. Además, será una realidad "física". De momento no nos preocuparemos por los detalles de esta sentencia.

Cada vez que creemos una instancia de una clase, estaremos creando un objeto que contiene su propia copia de cada variable de instancia definida por la clase. Por lo tanto, cada objeto **Caja** contendrá sus propias copias de las variables de instancia **ancho**, **alto** y **largo**. Para acceder a estas variables, utilizaremos el operador *punto* (.). El operador punto liga el nombre del objeto con el nombre de una de sus variables de instancia. Por ejemplo, la siguiente sentencia sirve para asignar a la variable **ancho** del objeto **miCaja** el valor l00.

```
miCaja.ancho = 100;
```

Esta sentencia indica al compilador que debe asignar a la copia de **ancho** que está contenida en el objeto **miCaja** el valor 100. En general, el operador punto se usa para acceder tanto a las variables como a los métodos de un objeto. El siguiente es un programa completo que utiliza la clase **Caja**:

```
/*  Un programa que utiliza la clase Caja.

    El nombre de este archivo es CajaDemo.java
*/
    class Caja {
    double ancho;
    double alto;
    double largo;
}

//  Esta clase declara un objeto de la clase Caja.
class CajaDemo {
  public static void main (String args[]) {
    Caja miCaja = new Caja();
    double vol;

    // asignación de valores a las variables del objeto miCaja
    miCaja.ancho = 10;
    miCaja.alto = 20;
    miCaja.largo = 15;

    // Se calcula el volumen de la caja
    vol = miCaja.ancho * miCaja.alto * miCaja.largo;

    System.out.println ("El volumen es " + vol);
  }
}
```

Al archivo que contiene este programa se le debe llamar **CajaDemo.java**, ya que el método **main( )** está dentro de la clase denominada **CajaDemo**, no en la clase denominada **Caja**. Cuando se compila este programa, se generan dos archivos **.class**, uno para **Caja** y otro para **CajaDemo**. El compilador Java crea automáticamente para cada clase su propio archivo **.class**. No es necesario que las clases **Caja** y **CajaDemo** estén en el mismo archivo fuente. Se puede escribir cada clase en su propio archivo, es decir, en los archivos **Caja.java** y **CajaDemo.java**, respectivamente.

Para ejecutar este programa, debemos ejecutar **CajaDemo.class**, y obtendremos la siguiente salida:

```
El volumen es 3000.0
```

Tal y como se ha visto anteriormente, cada objeto tiene sus propias copias de las variables de instancia. Esto significa que si tenemos dos objetos **Caja**, cada uno tiene sus propias copias de **largo**, **ancho** y **alto**. Es importante tener en cuenta que los cambios en las variables de instancia de un objeto no afectan a las variables de otro. Por ejemplo, el siguiente programa declara dos objetos **Caja**.

```
// Este programa declara dos objetos Caja.

class Caja {
  double ancho;
  double alto;
  double largo;
}

class CajaDemo2{
  public static void main (String args[]) {
    Caja miCajal = new Caja();
    Caja miCaja2 = new Caja();
    double vol;

    // asignación de valores a las variables de la instancia miCaja1
    miCaja1.ancho = 10;
    miCaja1.alto = 20;
    miCaja1.largo = 15;

    /* asignación de valores diferentes a las variables de la instancia miCaja2
    */
    miCaja2.ancho = 3;
    miCaja2.alto = 6;
    miCaja2.1argo = 9;

    // calcula el volumen de la primera caja
    vol = miCaja1.ancho * miCaja1.alto * miCaja1.largo;
    System.out.println("E1 volumen es " + vol);

    // calcula el volumen de la segunda caja
    vol = miCaja2.ancho * miCaja2.alto * miCaja2.largo;
    System.out.println("E1 volumen es " + vol);
  }
}
```

La salida que se obtiene es la siguiente:

```
El volumen es 3000.0
El volumen es 162.0
```

Como se puede comprobar, los datos de **miCajal** son completamente independientes de los datos contenidos en **miCaja2**.

Declaración de objetos

Tal y como se acaba de explicar, cuando se crea una clase, se está creando un nuevo tipo de datos que se utilizará para declarar objetos de ese tipo. Sin embargo, la obtención de objetos de una clase es un proceso que consta de dos etapas. En primer lugar, se debe declarar una variable del tipo de la clase. Esta variable no define un objeto, sino que simplemente es una *referencia* a un objeto. En segundo lugar, se debe obtener una copia física del objeto y asignarla a esa variable. Para ello se utiliza el operador **new** que asigna dinámicamente, durante el tiempo de ejecución, memoria a un objeto y devuelve una referencia al mismo. Esta referencia es algo así como la dirección en memoria del objeto creado por la operación **new**. Luego se almacena esta referencia en la variable. Todos los objetos de una clase en Java se asignan dinámicamente. Veamos con más detalle este procedimiento.

En los ejemplos anteriores se utilizó una línea similar a la siguiente para declarar un objeto de la clase **Caja**:

```
Caja miCaja = new Caja();
```

Esta sentencia combina las dos etapas descritas anteriormente y, para mostrar más claramente cada una de ellas, dicha sentencia se puede volver a escribir del siguiente modo:

```
Caja miCaja; // declara la referencia a un objeto
miCaja = new Caja(); // reserva espacio en memoria para el objeto
```

La primera línea declara **miCaja** como una referencia a un objeto de la clase **Caja**. Después de que se ejecute esta línea, **miCaja** contiene el valor **null**, que indica que todavía no apunta a un objeto real. Cualquier intento de utilizar **miCaja** en esta situación dará lugar a un error de compilación. En la siguiente línea se reserva memoria para un objeto real y se asigna **miCaja** como la referencia a dicho objeto. Una vez que se ejecute la segunda línea, ya se puede utilizar **miCaja** como si fuera un objeto de la clase **Caja**. En realidad, **miCaja** simplemente contiene la dirección de memoria del objeto real. El efecto de estas dos líneas se describe en la Figura 6.1.

NOTA *Los lectores familiarizados con C/C++ habrán observado, probablemente, que las referencias a objetos son muy semejantes a los apuntadores. Básicamente, esto es correcto. Una referencia a objeto es semejante a un apuntador a memoria. La principal diferencia –y la clave para la seguridad de Java– es que no se pueden manipular las referencias tal y como se hace con los apuntadores. Por lo tanto, una referencia no puede apuntar a una dirección arbitraria de memoria ni se puede manipular como si fuese entero.*

El operador new

Como se explicó, el operador **new** reserva memoria dinámicamente para un objeto. Su forma general es:

variable = new *nombre_de_clase* ();

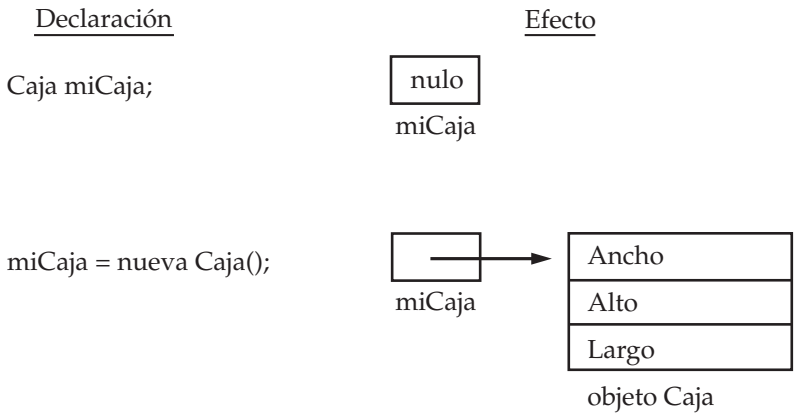


FIGURA 6-1
Declaración de un objeto de tipo **Caja.**

Aquí, *variable* es una variable cuyo tipo es la clase creada, y el nombre_de_clase es el nombre de la clase que está siendo instanciada. El *nombre de la clase* seguido de paréntesis está especificando una llamada al método constructor de la clase. Un *constructor* define lo que ocurre cuando se crea un objeto de una clase. Los constructores son una parte importante de todas las clases y tienen muchos atributos significativos. En la práctica, la mayoría de las clases definen explícitamente sus propios constructores en la definición de la clase. Cuando no se definen explícitamente, Java suministra automáticamente el constructor por omisión. Esto es lo que ha ocurrido con la clase **Caja**. Por ahora seguiremos utilizando el constructor por omisión, aunque pronto veremos cómo definir nuestros propios constructores.

En este momento nos podríamos plantear la siguiente pregunta: ¿Por qué no es necesario utilizar el operador **new** en el caso de los enteros o de los caracteres? La respuesta es que los tipos primitivos no se implementan como objetos sino como variables"normales". Esto se hace así con el objeto de lograr una mayor eficiencia. Los objetos tienen muchas características y atributos que obligan a Java a tratarlos de forma diferente a la que utiliza con los tipos básicos. Al no aplicar la misma sobrecarga a los tipos primitivos que a los objetos, Java puede implementar a los tipos básicos más eficientemente. Más adelante se verán versiones con objetos de los tipos primitivos, las cuales están disponibles para su uso en situaciones en las que se necesitan objetos completos para trabajar con valores primitivos.

Es importante tener en cuenta que el operador **new** reserva memoria para un objeto durante el tiempo de ejecución. La ventaja de hacerlo así es que el programa crea exactamente los objetos que necesita durante su ejecución. Sin embargo, dado que la memoria disponible es finita, puede ocurrir que ese operador **new** no sea capaz de reservar memoria para un objeto porque no exista ya memoria disponible. Si esto ocurre, se producirá una excepción en tiempo de ejecución. (En el Capítulo 10 se verá la gestión de ésta y otras excepciones). En los ejemplos que se presentan en este libro no es necesario que nos preocupemos por el hecho de quedamos sin memoria, pero sí es preciso considerar esta posibilidad en los programas reales.

Volvamos de nuevo a la distinción entre clase y objeto. Una clase crea un nuevo tipo de dato que se utilizará para crear objetos, es decir, una clase crea un marco lógico que define las relaciones entre sus miembros. Cuando se declara un objeto de una clase, se está creando una instancia de esa clase. Por lo tanto, una clase es una construcción lógica, mientras que un objeto

tiene una realidad física, esto es, un objeto ocupa un espacio de memoria. Es importante tener en cuenta esta distinción.

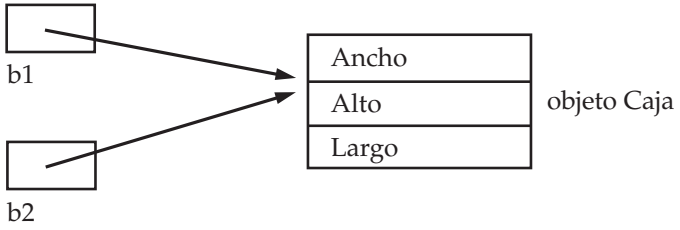
Asignación de variables de referencia a objetos

Las variables de referencia a objetos actúan de una forma diferente a la que se podría esperar cuando tiene lugar una asignación. Por ejemplo, ¿qué hace el siguiente fragmento de código?

```
Caja b1 = new Caja();
Caja b2 = b1;
```

Podríamos pensar que a **b2** se le asigna una referencia a una copia del objeto que se referencia mediante **b1**, es decir, que **b1** y **b2** se refieren a objetos distintos. Sin embargo, esto no es así. Cuando este fragmento de código se ejecute, **b1** y **b2** se referirán al mismo *objeto*. La asignación de **b1** a **b2** no reserva memoria ni copia parte alguna del objeto original. Simplemente hace que **b2** se refiera al mismo objeto que **b1**. Por lo tanto, cualquier cambio que se haga en el objeto a través de **b2** afectará al objeto al que se refiere **b1**, ya que, en definitiva, se trata del mismo objeto.

Esta situación se representa gráficamente a continuación.



Aunque **b1** y **b2** se refieren al mismo objeto, no están relacionados de ninguna otra forma. Por ejemplo, una asignación posterior a **b1** simplemente *desenganchará* **b1** del objeto original sin afectar al objeto o a **b2**. Por ejemplo:

```
Caja b1 = new Caja();
Caja b2 = b1;
// ...
b1 = null;
```

En este caso, **b1** ha sido asignado a **null**, pero **b2** todavía apunta al objeto original.

RECUERDE *Cuando se asigna una variable de referencia a objeto a otra variable de referencia a objeto, no se crea una copia del objeto, sino que sólo se hace una copia de la referencia.*

Métodos

Como se mencionó al comienzo de este capítulo, las clases están formadas por variables de instancia y métodos. El concepto de método es muy amplio ya que Java les concede una gran potencia y flexibilidad. La mayor parte del siguiente capítulo se dedica a los métodos. Sin

embargo, es preciso introducir en este momento algunas nociones básicas para empezar a incorporar métodos a las clases.

La forma general de un método es la siguiente:

tipo nombre_de_método (parámetros) {
  // cuerpo del método
}

Donde *tipo* especifica el tipo de dato que devuelve el método, el cual puede ser cualquier tipo válido, incluyendo los tipos definidos mediante clases creadas por el programador. Cuando el método no devuelve ningún valor, el tipo devuelto debe ser **void**. El nombre del método se especifica en *nombre_de_método*, que puede ser cualquier identificador válido que sea distinto de los que ya están siendo utilizados por otros elementos del programa. Los *parámetros* son una sucesión de pares de tipo e identificador separados por comas. Los parámetros son, esencialmente, variables que reciben los valores de los *argumentos* que se pasa a los métodos cuando se les llama. Si el método no tiene parámetros, la lista de parámetros estará vacía.

Los métodos que devuelven un tipo diferente del tipo **void** devuelven el valor a la rutina llamante mediante la siguiente forma de la sentencia **return**:

return *valor*;

Donde *valor* es el valor que el método retorna.

En los siguientes apartados se verá cómo crear distintos tipos de métodos, incluyendo los que tienen parámetros y los que devuelven valores.

Adición de un método a la clase Caja

Aunque crear una clase que contenga solamente datos es correcto, rara vez se hace. En la mayor parte de las ocasiones se usarán métodos para acceder a las variables de instancia definidas por la clase. De hecho los métodos definen la interfaz para la mayor parte de las clases. Esto permite que la clase oculte la estructura interna de los datos detrás de las abstracciones de un conjunto de métodos. Además de definir métodos que proporcionen el acceso a los datos, también se pueden definir métodos cuyo propósito sea el de ser utilizados internamente por la propia clase.

Comencemos por añadir un método a la clase **Caja**. En los programas anteriores se calculaba el volumen de una caja en la clase **CajaDemo**; sin embargo, el volumen de la caja depende del tamaño de la caja. Por este motivo tiene más sentido que sea la clase **Caja** la que se encargue del cálculo del volumen. Para ello se debe añadir un método a la clase **Caja**, tal y como se muestra a continuación:

```
// Este programa incluye un método en la clase Caja.

class Caja {
  double ancho;
  double alto;
  double largo;

  // presenta el volumen de una caja
  void volumen () {
    System.out.print ("El volumen es ");
    System.out.println (ancho * alto * largo);
  }
}
```

```
class CajaDemo3 {
  public static void main (String args[]) {
    Caja miCaja1 = new Caja();
    Caja miCaja2 = new Caja();

    // Se asignan valores a las variables del objeto miCaja1
    miCaja1.ancho = 10;
    miCaja1.alto = 20;
    miCaja1.largo = 15;

    /* asigna diferentes valores a las variables
       del objeto de miCaja2 */
    miCaja2.ancho = 3;
    miCaja2.alto = 6;
    miCaja2.largo = 9;

    // muestra el volumen de la primera caja
    miCaja1.volumen ();

    // muestra el volumen de la segunda caja
    miCaja2.volumen ();
  }
}
```

Este programa genera la siguiente salida, que es la misma que se obtuvo en la versión anterior.

```
El volumen es 3000.0
El volumen es 162.0
```

Analicemos más detenidamente las siguientes dos líneas de código:

```
miCaja1.volumen();
miCaja2.volumen();
```

La primera invoca al método **volumen( )** en **miCajal**, es decir, llama al método **volumen( )**, relativo al objeto **miCajal**, utilizando el nombre del objeto seguido por el operador punto. Por lo tanto, la llamada al método **miCaja1.volumen( )** presenta el volumen de la caja definida por **miCajal**, y la llamada a **miCaja2.volumen( )** presenta el volumen de la caja definida por **miCaja2**. Cada vez que se llama a **volumen( )** se presenta el volumen de la caja especificada.

Si no está familiarizado con el concepto de llamada a un método, el siguiente análisis le ayudará a aclarar las cosas. Cuando se ejecuta **miCaja1.volumen( )**, el intérprete de Java transfiere el control al código definido dentro del método **volumen( )**. Una vez que estas sentencias se han ejecutado, el control es devuelto a la rutina llamante, y la ejecución continúa en la línea de código que sigue a la llamada. En un sentido más general, un método de Java es una forma de implementar subrutinas.

Dentro del método **volumen( )**, es muy importante observar que la referencia a las variables de instancia **ancho**, **alto** y **largo** es directa sin que vayan precedidas del nombre de un objeto o del operador punto. Cuando un método utiliza una variable de instancia definida por su propia clase, lo hace directamente, sin referencia explícita a un objeto y sin utilizar el operador punto. Siempre que se llama a un método, esté está relacionado con algún objeto de su clase. Una vez que la llamada tiene lugar, el objeto es conocido.

Por lo tanto, en un método no es necesario especificar el objeto por segunda ocasión. Esto significa que **ancho**, **alto** y **largo** dentro de **volumen( )** se refieren implícitamente a las copias de esas variables que están en el objeto que llama a **volumen( )**.

Revisando, cuando se accede a una variable de instancia por un código que no forma parte de la clase en la que está definida la variable de instancia, se debe hacer mediante un objeto utilizando el operador punto. Sin embargo, cuando el código forma parte de la misma clase en la que se define la variable de instancia a la que accede dicho código, la referencia a esa variable puede ser directa. Esto se aplica de la misma forma a los métodos.

Devolución de un valor

La implementación del método **volumen( )** realiza el cálculo del volumen de una caja dentro de la clase **Caja** a la que pertenece, sin embargo esta implementación no es la mejor. Por ejemplo, puede ser un problema si en otra parte del programa se necesita el valor del volumen de la caja, pero sin que sea necesario presentar dicho valor. Una mejor forma de implementar el método **volumen( )** es realizar el cálculo del volumen y devolver el resultado a la parte del programa que llama al método. En el siguiente ejemplo, que es una versión mejorada del programa anterior, se hace eso.

```
// Ahora volumen() devuelve el volumen de una caja.

class Caja {
  double ancho;
  double alto;
  double largo;

  // cálculo y devolución del valor
  double volumen() {
    return ancho * alto * largo;
  }
}

class CajaDemo4 {
  public static void main (String args[]) {
    Caja miCaja1 = new Caja();
    Caja miCaja2 = new Caja();
    double vol;

    // se asigna valores a las variables de instancia de miCaja1
    miCaja1.ancho = 10;
    miCaja1.alto = 20;
    miCaja1.largo = 15;

    /* se asigna diferentes valores a las variables
       de instancia de miCaja2 */
    miCaja2.ancho = 3;
    miCaja2.alto = 6;
    miCaja2.largo = 9;

    // se obtiene el volumen de la primera caja
    vol = miCaja1. volumen ();
    System.out.println ("El volumen es " + vol);
```

```
    // se obtiene el volumen de la segunda caja
    vol = miCaja2 .volumen ();
    System.out.println ("El volumen es " + vol);
  }
}
```

En este ejemplo, cuando se llama al método **volumen( )**, se coloca en la parte derecha de la sentencia de asignación. En la parte izquierda está la variable, en este caso **vol**, que recibirá el valor devuelto por **volumen( )**. Por lo tanto, después de que se ejecute la sentencia:

```
vol = miCajal.volumen();
```

el valor de **miCajal.volumen( )** es 3,000 y este valor se almacena en vol.

Dos puntos importantes a considerar sobre la devolución de valores son:

- El tipo de datos devueltos por un método debe ser compatible con el tipo de retorno especificado por el método. Por ejemplo, si el tipo de retorno de un método es **booleano**, no se puede devolver un entero.
- La variable que recibe el valor devuelto por un método (vol, en este caso) debe ser también compatible con el tipo de retorno especificado por el método.

Una cuestión más: el programa anterior se puede escribir de forma más eficiente teniendo en cuenta que realmente no es necesario que exista la variable **vol**. Se puede utilizar la llamada a **volumen( )** directamente en la sentencia **println( )**, como se muestra a continuación.

```
System.out.println("El volumen es " + miCaja1.volumen());
```

En este caso, cuando se ejecuta **println( )**, se llama directamente a **miCajal.volumen( )** y se pasa su valor a **println( ).**

Métodos con parámetros

Mientras que algunos métodos no necesitan parámetros, la mayoría sí. Los parámetros permiten generalizar un método, es decir, un método con parámetros puede operar sobre gran variedad de datos y/o ser utilizado en un gran número de situaciones diferentes. Para ilustrar este punto usaremos un ejemplo muy sencillo. El siguiente método devuelve el cuadrado del número 10:

```
int cuadrado ()
{
  return 10 * 10;
}
```

Efectivamente este método devuelve el cuadrado de 10, pero su utilización es muy limitada. Sin embargo, si se modifica de forma que tome un parámetro, como se muestra a continuación, entonces se consigue que **cuadrado( )** tenga una mayor utilidad.

```
int cuadrado(int i)
{
  return i * i;
}
```

Ahora, **cuadrado( )** devolverá el cuadrado de cualquier valor usado en la llamada al método, es decir, **cuadrado( )** es ahora un método de propósito general que puede calcular el cuadrado de cualquier número entero.

Aquí está un ejemplo de ello:

```
int x, y;
x = cuadrado(5); // x es igual a 25
x = cuadrado(9); // x es igual a 81
y = 2;
x = cuadrado (y) ; // x es igual a 4
```

En la primera llamada a **cuadrado( )**, se pasa el valor 5 al parámetro **i**. En la segunda, **i** recibirá el valor 9. La tercera invocación pasa el valor de y, que en este ejemplo es 2. Como muestran estos ejemplos, **cuadrado( )** devuelve el cuadrado de cualquier valor que se pase al método.

Es importante tener una idea precisa de estos dos términos, *parámetros* y *argumentos*. Un *parámetro* es una variable, definida por un método, que recibe un valor cuando se llama al método. Por ejemplo, en **cuadrado( )** el parámetro es **i**. Un *argumento* es un valor que se pasa a un método cuando se le llama. Por ejemplo, **cuadrado(l00)** pasa 100 como un argumento. Dentro de **cuadrado( )**, el parámetro **i** recibe ese valor.

Se puede utilizar un método parametrizado para mejorar la clase **Caja**. En los ejemplos anteriores, las dimensiones de cada caja se establecen por separado mediante una sucesión de sentencias:

```
micaja1.ancho = 10;
miCaja1.alto = 20;
micaja1.largo = 15;
```

Este código funciona, pero presenta problemas por dos razones. En primer lugar, resulta torpe y propenso a errores; por ejemplo, fácilmente se puede olvidar dar valor a una de las dimensiones. En segundo lugar, en los programas de Java correctamente diseñados, sólo se puede acceder a las variables de instancia por medio de métodos definidos por sus clases. De ahora en adelante, permitiremos alterar el comportamiento de un método, pero no el de una variable de instancia accesible desde el exterior de la clase.

Una mejor solución es crear un método que tome las dimensiones de la caja dentro de sus parámetros y establezca las variables de instancia apropiadamente. En el siguiente programa se implementa este concepto:

```
// Este programa usa un método parametrizado.

class Caja {
  double ancho;
  double alto;
  double largo;

  // cálculo y devolución del volumen
  double volumen () {
    return ancho * alto * largo;
  }

  // establece las dimensiones de la caja
  void setDim (double w, double h, double d) {
    ancho = w;
```

```
    alto = h;
    largo = d;
  }
}

class CajaDemo5 {
  public static void main (String args[]) {
    Caja miCaja1 = new Caja();
    Caja miCaja2 = new Caja();
    double vol;

    // inicializa cada caja
    miCaja1.setDim (10, 20, 15);
    miCaja2.setDim (3, 6, 9);

    // calcula el volumen de la primera caja
    vol = miCaja1.volumen ();
    System.out.println ("El volumen es " + vol);

    // calcula el volumen de la segunda caja
    vol = miCaja2.volumen ();
    System.out.println ("El volumen es " + vol);
  }
}
```

El método **setDim( )** se utiliza para establecer las dimensiones de cada caja. Por ejemplo, cuando se ejecuta:

```
miCaja1.setDim(10, 20, 15);
```

el valor 10 se copia en el parámetro **w**; el valor 20, en el parámetro **h**, y el valor 15, en el parámetro **d**. Dentro del método **setDim( )** los valores de **w**, **h** y **d** se asignan a las variables **ancho**, **alto** y **largo**, respectivamente.

Para muchos lectores, los conceptos presentados en los apartados anteriores les resultarán familiares. Sin embargo, si conceptos tales como la llamada a métodos, argumentos y parámetros le resultan nuevos, puede resultar conveniente que dedique algún tiempo a familiarizarse con ellos antes de seguir adelante, puesto que son fundamentales para la programación en Java.

Constructores

El proceso de inicializar todas las variables en una clase cada vez que se crea una instancia puede resultar tedioso, incluso cuando se añaden métodos como **setDim( )**. Puede resultar más simple y más conciso realizar todas las inicializaciones cuando el objeto se crea por primera vez. El proceso de inicialización es tan común que Java permite que los objetos se inicialicen cuando son creados. Esta inicialización automática se lleva a cabo mediante el uso de un constructor.

Un *constructor* inicializa un objeto inmediatamente después de su creación. Tiene el mismo nombre que la clase en la que reside y, sintácticamente, es similar a un método. Una vez definido, se llama automáticamente al constructor después de crear el objeto y antes de que termine el operador **new**. Los constructores resultan un poco diferentes, a los métodos convencionales, porque no devuelven ningún tipo, ni siquiera **void**. Esto se debe a que el tipo implícito que devuelve un constructor de clase es el propio tipo de la clase. La tarea del constructor es inicializar el estado interno de un objeto de forma que el código que crea a la

instancia pueda contar con un objeto completamente inicializado que pueda ser utilizado inmediatamente.

Se puede modificar el ejemplo anterior de forma que las dimensiones de la caja se inicialicen automáticamente cuando se construye el objeto. Para ello se sustituye el método **setDim( )** por un constructor. Comencemos definiendo un constructor sencillo que simplemente asigne los mismos valores a las dimensiones de cada caja.

```
/* La clase Caja usa un constructor para inicializar
   las dimensiones de las caja.
*/
class Caja {
  double ancho;
  double alto;
  double largo;

  // Este es el constructor para Caja.
  Caja() {
    System.out.println("Constructor de Caja");
    ancho = 10;
    alto = 10;
    largo = 10;
  }

  // calcula y devuelve el volumen
  double volumen () {
    return ancho * alto * largo;
  }
}

class CajaDemo6 {
  public static void main (String args[]) {
    // declara, reserva memoria, e inicial iza objetos de tipo Caja
    Caja miCaja1 = new Caja();
    Caja miCaja2 = new Caja();

    double vol;

    // obtiene el volumen de la primera caja
    vol = miCaja1.volumen () ;
    System.out.println ("El volumen es " + vol);

    // obtiene el volumen de la segunda caja
    vol = miCaja2.volumen ();
    System.out.println ("El volumen es " + vol);
  }
}
```

Cuando se ejecuta este programa, genera el siguiente resultado:

```
Constructor de Caja
Constructor de Caja
El volumen es 1000.0
El volumen es 1000.0
```

Como puede observarse, **miCaja1** y **miCaja2** han sido inicializados por el constructor de **Caja( )** en el momento de su creación. Como el constructor asigna el mismo valor, 10, a

todas las dimensiones de la caja, **miCajal** y **miCaja2** tienen el mismo volumen. La sentencia **println( )** dentro de **Caja( )** sólo sirve para mostrar cómo funciona el constructor. La mayoría de los constructores no presentan alguna salida, sino que simplemente inicializan un objeto.

Antes de seguir, examinemos de nuevo el operador **new**. Cuando se reserva espacio de memoria para un objeto, se hace de la siguiente forma:

variable = new *nombre_de_clase ();*

Ahora resulta más evidente la necesidad de los paréntesis después del nombre de clase. Lo que ocurre realmente es que se está llamando al constructor de la clase. Por lo tanto, en la línea:

```
Caja miCajal = new Caja();
```

new Caja( ) es la llamada al constructor de **Caja( )**. Cuando no se define explícitamente un constructor de clase, Java crea un constructor por defecto de clase. Este es el motivo de que la línea anterior funcionara correctamente en las versiones previas de **Caja** en las que no se definía constructor alguno. El constructor por omisión asigna, automáticamente, a todas las variables el valor inicial igual a cero. Para clases sencillas, resulta suficiente utilizar el constructor por defecto, pero no para clases más sofisticadas. Una vez definido el propio constructor, el constructor por omisión ya no se utiliza.

Constructores con parámetros

Aunque el constructor de **Caja( )** en los ejemplos previos inicializa un objeto **Caja**, no es muy útil que todas las cajas tengan las mismas dimensiones. Necesitamos una forma de construir objetos **Caja** de diferentes dimensiones. La solución más sencilla es añadir parámetros al constructor, con lo que se consigue que éste sea mucho más útil. La siguiente versión de **Caja** define un constructor con parámetros que asigna a las dimensiones de la caja los valores especificados por esos parámetros.

Prestemos especial atención a la forma en que se crean los objetos de **Caja**.

```
/* Aquí, Caja usa un constructor parametrizado para
   inicializar las dimensiones de una caja.
*/
class Caja {
  double ancho;
  double alto;
  double largo;

  // Este es el constructor de Caja.
  Caja (double w, double h, double d) {
    ancho = w;
    alto = h;
    largo = d;
  }

  // calcula y devuelve el volumen
  double volumen () {
    return ancho * alto * largo;
  }
}

class CajaDemo7 {
  public static void main(String args[]) {
```

```
    // declara, reserva memoria, e inicializa los objetos de Caja
    Caja miCaja1 = new Caja(10, 20, 15);
    Caja miCaja2 = new Caja(3, 6, 9);

    double vol;

    // obtiene el volumen de la primera caja
    vol = miCaja1.volumen();
    System.out.println ("El volumen es " + vol);

    // obtiene el volumen de la segunda caja
    vol = miCaja2.volumen();
    System.out.println ("El volumen es " + vol);
  }
}
```

La salida de este programa es la siguiente:

```
El volumen es 3000.0
El volumen es 162.0
```

Como se puede ver, cada objeto es inicializado como se especifica en los parámetros de su constructor. Por ejemplo, en la siguiente línea:

```
Caja miCaja1 = new Caja(10, 20, 15);
```

Los valores 10, 20 y 15 se pasan al constructor de **Caja( )** cuando **new** crea el objeto. Así las copias de **ancho**, **alto** y **largo** de **miCaja1** contendrán los valores 10, 20 y 15, respectivamente.

La palabra clave this

En algunas ocasiones, un método necesita referirse al objeto que lo invocó. Para permitir esta situación, Java define la palabra clave **this**, la cual puede ser utilizada dentro de cualquier método para referirse al objeto actual. **this** es siempre una referencia al objeto sobre el que ha sido llamado el método. Se puede usar **this** en cualquier lugar donde esté permitida una referencia a un objeto del mismo tipo de la clase actual.

Consideremos la siguiente versión de **Caja( )** para comprender mejor cómo funciona **this**.

```
// Un uso redundante de this.
Caja (double w, double h, double d) {
  this.ancho = w;
  this.alto = h;
  this.largo = d;
}
```

Esta versión de **Caja( )** opera exactamente igual que la versión anterior. El uso de **this** es redundante pero correcto. Dentro de **Caja( )**, **this** se refiere siempre al objeto llamante. Aunque en este caso es redundante, en otros contextos **this** es útil; uno de esos contextos se explica en la siguiente sección.

Ocultando variables de instancia

En Java es ilegal declarar a variables locales con el mismo nombre dentro del mismo contexto. Curiosamente, puede haber variables locales, desde parámetros formales hasta métodos, que coincidan en parte con los nombres de las variables de instancia de clase. Sin embargo, cuando

una variable tiene el mismo nombre que una variable de instancia, la variable local *esconde* a la variable de instancia. Por esta razón, **ancho**, **alto** y **largo** no se utilizaron como los nombres de los parámetros en el constructor **Caja( )** dentro de la clase **Caja**. Si se hubieran utilizado, entonces **ancho** se hubiera referido al parámetro formal, ocultando la variable de instancia ancho. Si bien normalmente será más sencillo utilizar nombres diferentes, **this** permite hacer referencia directamente al objeto y resolver de esta forma cualquier colisión entre nombres, que pudiera darse entre las variables de instancia y las variables locales. La siguiente versión de **Caja( )** utiliza **ancho**, **alto**, y **largo** como nombres de parámetros y, después, **this** para acceder a variables de instancia que tienen los mismos nombres.

```
// Uso de this para resolver colisiones en el espacio de nombres
Caja (double ancho, double alto, double largo) {
  this.ancho = ancho;
  this.alto = alto;
  this.largo = largo;
}
```

NOTA *El uso de **this** en este contexto puede ser confuso, y algunos programadores tienen la precaución de no utilizar nombres de variables locales y parámetros formales que puedan ocultar variables de instancia. Otros programadores creen precisamente lo contrario, es decir, que puede resultar conveniente, para una mayor claridad, utilizar los mismos nombres, y usan **this** para superar el ocultamiento de la variable de instancia. Adoptar una tendencia u otra es una cuestión de preferencias.*

Recolección automática de basura

Ya que en Java se reserva espacio de memoria para los objetos dinámicamente mediante la utilización de operador **new**, surge la pregunta sobre cómo destruir los objetos y liberar el correspondiente espacio de memoria para su posterior utilización. En algunos lenguajes como C++, la memoria asignada dinámicamente debe ser liberada de forma manual mediante el operador **delete**. Esta situación se resuelve en Java de forma diferente. Java gestiona automáticamente la liberación de la memoria. Esta técnica se denomina *recolección de basura* y consiste en lo siguiente: cuando no existen referencias a un objeto, se asume que el objeto no se va a necesitar más, y la memoria ocupada por dicho objeto puede ser liberada. No es necesario destruir objetos explícitamente como en C++. La recolección de basura sólo se produce esporádicamente durante la ejecución del programa. No se producirá simplemente porque haya uno o dos objetos que no se utilicen más. Los diferentes intérpretes de Java siguen distintos procedimientos de recolección de basura, pero en realidad no hay que preocuparse mucho por ello al escribir nuestros programas.

El método finalize( )

En algunas ocasiones es necesario realizar alguna acción cuando se destruye un objeto. Por ejemplo, si un objeto sustenta algún recurso que no pertenece a Java, como un descriptor de archivo o un tipo de letra del sistema de ventanas, entonces es necesario liberar estos recursos antes de destruir el objeto.

Para gestionar estas situaciones, Java proporciona un mecanismo denominado *finalización* mediante el cual se pueden definir acciones específicas que se producirán cuando el sistema de recolección de basura vaya a eliminar un objeto.

Para añadir un finalizador a una clase basta con definir el método **finalize( )**. El intérprete de Java llamará a ese método siempre que esté a punto de eliminar un objeto de esa clase. Dentro del método **finalize( )** se especificarán aquellas acciones que se han de efectuar antes de destruir un objeto. El sistema de recolección de basura se ejecuta periódicamente, buscando objetos a los que ya no haga referencia ningún estado en ejecución, o indirectamente a través de otros objetos referenciados. Justo antes de eliminar un objeto, el intérprete de Java llama al método **finalize( )** de ese objeto.

El método **finalize( )** tiene la forma general:

```
protected void finalize ( )
{
// código de finalización
}
```

Aquí, la palabra clave **protected** es un especificador que impide el acceso a **finalize( )** por parte de un código definido fuera de su clase. Éste y otros especificadores de acceso se explican en el Capítulo 7.

Es importante entender que sólo se llama al método **finalize( )** justo antes de que actúe el sistema de recolección de basura, y no, por ejemplo, cuando un objeto está fuera del contexto. Esto significa que no se puede saber exactamente cuándo será, o incluso si será, ejecutado el método **finalize( ).** Por lo tanto, el programa debe incluir otros medios que permitan liberar los recursos del sistema y anexos utilizados por el objeto. No nos debemos apoyar en el método **finalize( )** para la operación normal del programa.

NOTA *Si usted está familiarizado con C++ entonces sabe que C++ permite la definición de un destructor para una clase al que se llama cuando un objeto queda fuera de contexto. Java no proporciona destructores basándose en este concepto. El método* ***finalize( )*** *consiste únicamente en un aproximación a esta funcionalidad. A medida que usted vaya adquiriendo una mayor experiencia en el manejo de Java, verá que la necesidad de las funciones de un destructor es mínima, debido al sistema de recolección de basura de que dispone Java.*

Una clase Stack

Aunque la clase **Caja** ha sido útil para ilustrar los elementos esenciales de una clase, su valor práctico es escaso. Este capítulo termina con un ejemplo más sofisticado que permite mostrar la verdadera potencia de las clases. Como recordará del análisis sobre programación orientada a objetos (POO), presentado en el Capítulo 2, una de las ventajas más importantes de la misma es el encapsulado de datos y código. La clase es el mecanismo por medio del cual se consigue dicho encapsulado en Java. Al crear una clase, se crea un nuevo tipo de datos que definen tanto la naturaleza de los datos como las rutinas utilizadas para manipularlos. Además, los métodos definen un interfaz consistente y controlada para los datos de la clase. Por lo tanto, se puede utilizar la clase a través de sus métodos sin preocuparse por los detalles de su implementación o por la gestión real de los datos dentro de la clase. En cierto sentido, una clase es como"una caja

negra". No es necesario saber lo que ocurre dentro de la caja para poder utilizarla por medio de su interfaz. De hecho al estar oculto el contenido de la "caja" éste puede cambiar sin afectar la percepción exterior. A medida que nuestro código utiliza a la clase por medio de sus métodos, los detalles internos pueden cambiar sin causar efectos fuera de la clase.

La aplicación práctica de lo dicho anteriormente se muestra mediante uno de los ejemplos típicos del encapsulamiento: la pila. Una *pila* (*stack*, por su nombre en inglés) almacena datos de manera que se retiran en orden inverso al de entrada, es decir, un stack es como una pila de platos encima de una mesa el primer plato puesto encima de la mesa es el último en ser utilizado. Las pilas se controlan mediante dos operaciones tradicionales denominadas *push* y *pop*. Para colocar un dato en la parte superior de la pila se utiliza la operación de push, y para retirarlo la operación pop. Veamos cuan sencillo resulta encapsular el mecanismo completo de una pila.

La clase denominada **Stack**, que se muestra a continuación, implementa una pila de enteros.

```
// Esta clase define un pila de enteros que puede almacenar hasta 10 valores.
class Stack {
  int stck[] = new int[10];
  int tos;

  // Inicializa el índice del elementos superior en la pila
  Stack () {
    tos = -1;
  }

  // Coloca un dato en la pila
  void push (int item) {
    if (tos == 9)
      System.out.println("La pila está llena.");
    else
      stck[++tos] = item;
  }

  // Retira un dato de la pila
  int pop () {
    if (tos < 0) {
      System.out.println("La pila está vacía.");
      return 0;
    }
    else
      return stck [tos--];
  }
}
```

La clase **Stack** define dos variables y tres métodos. El arreglo **stck** almacena la pila de enteros.

Este arreglo es indexado por la variable **tos**, que contiene en todo momento el índice del elemento en la parte superior de la pila. El constructor **Stack( )** inicializa la variable **tos** con el valor –1, que indica que la pila está vacía. El método **push( )** coloca un dato en la pila, y para recuperarlo se llama al método **pop( )**. Como el acceso a la pila se lleva a cabo mediante los métodos **push( )** y **pop( ),** el hecho de que la pila esté almacenada en un vector no tiene importancia por lo que se refiere a la utilización de la pila. Por ejemplo, aunque una pila pueda estar almacenada en una estructura de datos más compleja como una lista, la interfaz definida por **push( )** y **pop( )** será la misma.

La clase **TestStack** prueba el funcionamiento de la clase **Stack**: crea dos pilas de enteros, coloca algunos valores en cada una y después los retira:

```
class TestStack {
  public static void main (String args[]) {
    Stack miPilal = new Stack();
    Stack miPila2 = new Stack();

    // pone algunos números en la pila
    for (int i=0; i<l0; i++) miPilal.push(i);
    for (int i=l0; i<20; i++) miPila2.push(i);

    // retira esos números de la pila
    System.out.println ("Contenido de miPilal:");
    for (int i=0; i<l0; i++)
      System.out.println ( miPilal.pop() );

    System.out.println ("contenido de miPila2:");
    for (int i=0; i<l0; i++)
      System.out.println ( miPila2.pop() );
  }
}
```

Este programa genera la siguiente salida:

```
Contenido de miPilal:
9
8
7
6
5
4
3
2
1
0
Contenido de miPila2:
19
18
17
16
15
14
13
12
11
10
```

Como se puede comprobar, los contenidos de las dos pilas son totalmente independientes.

Por último, tal y como se ha implementado la clase **Stack** es posible que un código no perteneciente a la clase modifique el arreglo **stck** que almacena los valores de la pila. Esto podría ocasionar un uso o comportamiento incorrecto de la clase **Stack**. En el próximo capítulo se presenta la solución de este problema.

CAPÍTULO 7

Métodos y clases

Este capítulo continúa la discusión sobre los métodos y clases iniciada en el capítulo anterior. Se examinan varios temas relativos a los métodos que incluyen la sobrecarga, paso de parámetros y recursividad. Además se retoma el tema de las clases, discutiendo el control de acceso, el uso de la palabra clave **static**, y una de las clases más importantes que incorpora Java: **String**.

Sobrecarga de métodos

En Java es posible definir dos o más métodos que compartan el mismo nombre, dentro de la misma clase siempre y cuando la declaración de sus parámetros sea diferente. Cuando se produce esta situación se dice que los métodos están *sobrecargados,* y que el proceso es llamado *sobrecarga de métodos.* La sobrecarga de métodos es una de las formas en que Java implementa el polimorfismo. Si nunca ha utilizado un lenguaje que permita la sobrecarga de métodos, entonces este concepto puede resultar extraño en principio, pero tal y como se verá, la sobrecarga de métodos es una de las características más útiles e interesantes de Java.

Cuando se invoca a un método sobrecargado, Java utiliza el tipo y/o el número de argumentos como guía para determinar a qué versión del método sobrecargado se debe llamar. Por lo tanto, los métodos sobrecargados deben diferir en el tipo y/o número de sus parámetros. Mientras que los métodos sobrecargados pueden tener diferente tipo de retorno, el tipo de retorno por sí solo es insuficiente para distinguir entre dos versiones de un método. Cuando Java encuentra una llamada a un método sobrecargado, ejecuta la versión del método cuyos parámetros coinciden con los argumentos utilizados en la llamada.

A continuación se presenta un ejemplo que ilustra la sobrecarga de métodos.

```
// Ejemplo de sobrecarga de métodos.
class OverloadDemo {
  void test() {
    System.out.println("Sin parámetros");
  }

  // Sobrecarga el método test con un parámetro entero.
  void test(int a) {
    System.out.println("a: " + a);
  }
```

```
  // Sobrecarga el método test con dos parámetros enteros.
  void test(int a, int b) {
    System.out.println("a y b: " + a + " " + b);
  }

  // Sobrecarga el método test con un parámetro doble
  double test(double a) {
    System.out.println("a double: " + a);
    return a*a;
  }
}

class Overload {
  public static void main(String args[]) {
    OverloadDemo ob = new OverloadDemo();
    double result;

    // llamada a todas las versiones del método test()
    ob.test();
    ob.test(10);
    ob.test(10, 20);
    result = ob.test(123.25);
    System.out.println("Resultado de ob.test(123.25): " + result);
  }
}
```

Este programa genera la siguiente salida

```
Sin parámetros
a: 10
a y b: 10 20
a double: 123.25
Resultado de ob.test(123.25) : 15190.5625
```

En este ejemplo, el método **test( )** se sobrecarga cuatro veces. La primera versión no tiene parámetros, la segunda tiene un parámetro entero, la tercera dos parámetros enteros y la cuarta un parámetro **double**. El hecho de que la cuarta versión de **test( )** también devuelva un valor no tiene relación con la sobrecarga, ya que los tipos devueltos no desempeñan ningún papel en la resolución de la sobrecarga.

Cuando se llama a un método sobrecargado, Java busca las coincidencias entre los argumentos utilizados en la llamada y los parámetros del método. Sin embargo, esta coincidencia no es preciso que siempre sea exacta. En algunos casos se puede aplicar la conversión automática de tipos de Java. Por ejemplo, consideremos el siguiente programa:

```
// Aplicación de la conversión automática de tipos en la sobrecarga.
class OverloadDemo {
  void test () {
    System.out.println("Sin parámetros");
  }

  // Sobrecarga del método test con dos parámetros enteros.
  void test(int a, int b) {
    System.out.println("a y b: " + a + " " + b);
  }
```

```
  // Sobrecarga del método test con un parámetro double.
  void test(double a) {
    System.out.println("Dentro de test (double) a: " + a);
  }
}

class Overload {
  public static void main(String args[]) {
    OverloadDemo ob =new OverloadDemo();
    int i = 88;

    ob.test ();
    ob.test(l0, 20);

    ob.test(i): // esto llama a test (double)
    ob.test(123.2); // esto llama a test (double)
  }
}
```

Este programa genera la siguiente salida:

```
Sin parámetros
a y b: 10 20
Dentro de test (double) a: 88
Dentro de test (double) a: 123.2
```

Como puede verse, esta versión de **OverloadDemo** no define **test(int)** y cuando se llama a **test( )** con un argumento entero dentro de **Overload** no existe ningún método que coincidencia con esos parámetros. Sin embargo, Java puede convertir un entero en **double**, y esta conversión se puede utilizar para resolver la llamada. Por este motivo, cuando no se encuentra el método **test(int)**, Java eleva **i** a **double** y entonces llama a **test(double).** Naturalmente, si se hubiera definido **test(int)**, este método hubiera sido el llamado en lugar de **test(double)**. Java emplea su conversión automática de tipos sólo en el caso de no encontrar una coincidencia exacta.

El polimorfismo se basa en la sobrecarga de métodos, ya que es una de las formas en que Java implementa el paradigma de"una interfaz, múltiples métodos". Para comprender este concepto consideremos lo siguiente. En los lenguajes que no disponen de la sobrecarga de métodos, se debe dar un nombre único a cada método. Sin embargo, con frecuencia, se deseará implementar el mismo método para diferentes tipos de datos. Por ejemplo, la función valor absoluto, en los lenguajes que no disponen de la sobrecarga de métodos hay normalmente tres o más versiones de esta función, cada una de ellas con un nombre ligeramente distinto. Por ejemplo, en C, la función **abs( )** devuelve el valor absoluto de un entero, **labs( )** devuelve el valor absoluto de un entero largo, y **fabs( )** devuelve el valor absoluto de un valor de punto flotante. Como C no dispone de la sobrecarga de métodos, cada una de estas funciones ha de tener su propio nombre, aunque las tres hacen esencialmente lo mismo, y esto da lugar a una situación más compleja, conceptualmente, de lo que es en realidad. Aunque el concepto que subyace bajo cada una de estas funciones es el mismo, es necesario recordar tres nombres. Esta situación no se produce en Java porque cada método para obtener el valor absoluto puede utilizar el mismo nombre.

En efecto, la biblioteca estándar de Java incluye un método para obtener el valor absoluto, llamado **abs( ).** Este método sobrecargado de la clase **Math** puede gestionar cualquier tipo de dato numérico. Java determina a qué versión de **abs( )** se debe llamar según el tipo de argumentos.

La importancia de la sobrecarga de métodos es que permite relacionar los métodos a los que se va a acceder mediante el uso de un nombre común. Así, el nombre **abs** representa la *acción general* que se va a llevar a cabo. Queda para el compilador la elección de una versión *específica* adecuada a la circunstancia particular. El programador sólo necesita recordar la operación general que se quiere realizar. Mediante la aplicación del polimorfismo, varios nombres se han reducido a uno. Aunque este ejemplo es bastante sencillo, si se generaliza el concepto, resulta evidente la ayuda que supone la sobrecarga en el manejo de situaciones más complejas.

Cuando se sobrecarga un método, cada versión de ese método puede realizar cualquier actividad deseada. No existe una regla que establezca que cada método sobrecargado deba relacionarse con los demás. Sin embargo, desde el punto de vista del estilo, la sobrecarga de métodos implica una relación. Aunque se puede utilizar el mismo nombre para sobrecargar métodos que no están relacionados, no se debe hacer. Por ejemplo, se puede utilizar el nombre **sqr** para crear métodos que devuelvan el *cuadrado* de un entero y la *raíz cuadrada* de un número de punto flotante, pero estas dos operaciones son fundamentalmente distintas y esto va en contra del propósito original de la sobrecarga de métodos. En la práctica sólo se debe sobrecargar operaciones estrechamente relacionadas.

Sobrecarga de constructores

Además de sobrecargar métodos normales, también se puede sobrecargar métodos constructores. Para las clases que se crean en la práctica, la sobrecarga de métodos constructores será la norma y no la excepción. Para entender esto, volvamos a la clase **Caja** del capítulo anterior. A continuación se presenta la última versión de la clase **Caja**:

```
class Caja {
  double ancho;
  double alto;
  double largo;

  // Este es el constructor para Caja.
  Caja(double w, double h, double d) {
    ancho = w;
    alto = h;
    largo = d;
  }

  // cálculo y devolución del volumen
  double volumen() {
    return ancho * alto * largo;
  }
}
```

El constructor de **Caja( )** requiere tres parámetros. Esto significa que todas las declaraciones de objetos **Caja** deben pasar tres argumentos al constructor de **Caja**. De forma que la siguiente sentencia no es válida:

```
Caja ob = new Caja();
```

Como **Caja( )** necesita tres argumentos, es un error llamar a su método constructor sin dichos parámetros. Esto plantea algunas cuestiones importantes, por ejemplo, en el caso de que simplemente se quiera una caja y no importen sus dimensiones iniciales, o en el caso de que

se quiera inicializar un cubo especificando un único valor que debe ser utilizado para las tres dimensiones. Tal y como está escrita la clase **Caja**, estas opciones no son posibles.

Afortunadamente, la solución es sencilla y consiste simplemente en sobrecargar el constructor de **Caja** de forma que pueda abordar situaciones como las que se acaban de describir. El siguiente programa muestra una versión mejorada de la clase **Caja** que realiza sobrecarga del método constructor.

```
/* Aquí Caja define tres constructores que inicializan
   las dimensiones de la caja de varias formas.
*/
class Caja (
  double ancho;
  double alto;
  double largo;

  // constructor que se utiliza cuando se especifican
     todas las dimensiones
  Caja(double w, double h, double d) {
    ancho = w;
    alto = h;
    largo = d;
  }

  // constructor que se utiliza cuando
     no se especifican dimensiones
  Caja () {
    ancho = -1; // usa -1 para indicar
    alto = -1; //  una caja que no a sido
    largo = -1; // inicializada
  }

  // constructor que se utiliza para crear un cubo
  Caja(double lado) {
    ancho = alto = largo = lado;
  }

  // calcula y devuelve el volumen
  double volumen() {
    return ancho * alto * largo;
  }
}

class OverloadCons {
  public static void main(String args[]) {
    // crea cajas utilizando los distintos constructores
    Caja miCaja1 = new Caja(10, 20, 15);
    Caja miCaja2 = new Caja();
    Caja miCubo = new Caja(7);

    double vol;

    // se obtiene el volumen de la primera caja
    vol = miCaja1.volumen();
    System.out.println("El volumen de miCaja1 es " + vol);
```

```
    // se obtiene el volumen de la segunda caja
    vol = miCaja2.volumen();
    System.out.println("El volumen de miCaja2 es " + vol);
    // se obtiene el volumen del cubo
    vol = miCubo.volumen();
    System.out.println("El volumen de miCubo es " + vol);
  }
}
```

La salida que produce este programa es:

```
El volumen de miCajal es 3000.0
El volumen de miCaja2 es -1.0
El volumen de miCubo es 343.0
```

Como se ve, el constructor adecuado es llamado acorde a los parámetros especificados en la ejecución del operador **new**.

Uso de objetos como parámetros

Hasta el momento sólo hemos utilizado tipos simples como parámetros de los métodos. Sin embargo, también es habitual pasar objetos a los métodos. Considere, por ejemplo, el siguiente programa:

```
// Ejemplo de paso de objetos a los métodos.
class Test {
  int a, b;

  Test(int i, int j) {
    a = i;
    b = j;
  }

  // devuelve el valor verdadero si "o" es igual que el objeto
     que llama al método
  boolean equals(Test o) {
    if(o.a == a && o.b == b) return true;
    else return false;
  }
}

class PassOb {
  public static void main(String args[]) {
    Test ob1 = new Test(100, 22);
    Test ob2 = new Test(100, 22);
    Test ob3 = new Test(-1, -1);

    System.out.println("ob1 == ob2: " + ob1.equals(ob2));
    System.out.println("ob1 == ob3: " + ob1.equals(ob3));
  }
}
```

Este programa genera la siguiente salida:

```
ob1 == ob2: true
ob1 == ob3: false
```

El método **equals( )** dentro de **Test** compara la igualdad de dos objetos y devuelve el resultado, es decir, compara el objeto llamante con el que figura como argumento del método. Si contienen los mismos valores, el método devuelve el valor **true**. En caso contrario, devuelve el valor **false**. El parámetro **o** en **equals( )** especifica que su tipo es **Test**. Aunque **Test** es un tipo de clase creada por el programa, se utiliza de la misma forma que los tipos que Java tiene incorporados.

Uno de los usos más comunes de los objetos como parámetros es precisamente la que se da en los constructores. Con frecuencia se desea construir un objeto nuevo que sea inicialmente igual a otro objeto que ya existe. Para ello se debe definir un constructor que tome un objeto de su clase como parámetro. La siguiente versión de **Caja** permite que un objeto se tome para inicializar otro:

```
// Este ejemplo muestra como un objeto se utiliza para inicializar a otro.

class Caja {
  double ancho;
  double alto;
  double largo;

  // Observe que este constructor recibe como parámetro un objeto de tipo Caja
  Caja(Caja ob) { // se pasa el objeto al constructor
    ancho = ob.ancho;
    alto = ob.alto;
    largo = ob.largo;
  }

  // constructor que se utiliza cuando se especifican todas las dimensiones
  Caja(double w, double h, double d) {
    ancho = w;
    alto = h;
    largo = d;
  }

  // constructor que se utiliza cuando no se especifican dimensiones
  Caja () {
    ancho = -1; // usa -1 para indicar
    alto = -1; //  una caja que no ha sido
    largo = -1; // inicializada
  }

  // constructor que se utiliza para crear un cubo
  Caja(double lado) {
    ancho = alto = largo = lado;
  }

  // cálculo y devolución del volumen
  double volumen() {
    return ancho * alto * largo;
  }
}

class OverloadCons2 {
  public static void main(String args[]) {
```

```
    // se crean cajas usando los diferentes constructores
    Caja miCaja1 = new Caja(10, 20, 15);
    Caja miCaja2 = new Caja();
    Caja miCubo = new Caja(7);

    Caja miClon = new Caja(miCaja1); // crea una copia del objeto miCaja1

    double vol;

    // se obtiene el volumen de la primera caja
    vol = miCaja1.volumen();
    System.out.println("El volumen de miCaja1 es " + vol);
    // se obtiene el volumen de la segunda caja
    vol = miCaja2.volumen();
    System.out.println("El volumen de miCaja2 es " + vol);

    // se obtiene el volumen del cubo
    vol = miCubo.volumen();
    System.out.println("El volumen del cubo es " + vol);

    // se obtiene el volumen del clon
    vol = miClon.volumen();
    System.out.println("El volume del clon es " + vol);
  }
}
```

Cuando se comienza a crear clases propias, normalmente es preciso proporcionar varios métodos constructores que permitan la construcción de objetos de una manera conveniente y eficaz.

Paso de argumentos

En general, existen dos formas en las que un lenguaje de programación puede pasar un argumento a una subrutina. La primera es la de *llamada por valor*. Este método consiste en copiar el *valor* del argumento en el parámetro formal de la subrutina. Los cambios hechos en el parámetro de la subrutina no tienen efecto en el argumento usado para la llamada. La segunda forma en la que se puede pasar un argumento es la *llamada por referencia*. En este método, lo que se pasa es la referencia al argumento, no el valor del argumento. Dentro de la subrutina, esta referencia se usa para acceder al argumento que está realmente especificado en la llamada. Esto significa que los cambios realizados en el parámetro afectarán al argumento utilizado para llamar a la subrutina. En Java se utilizan ambos métodos en función del parámetro que se pasa a la subrutina.

En Java, cuando se pasa un tipo simple a un método se pasa por valor. De esta forma, lo que ocurra con el parámetro que recibe el argumento no tiene efecto alguno fuera del método. Considere, por ejemplo, el siguiente programa:

```
// Los tipos simples se pasan por valor.
class Test {
  void meth(int i, int j) {
    i *= 2;
    j /= 2;
  }
}
```

```
class LlamadaPorValor {
  public static void main(String args[]) {
    Test ob = new Test();

    int a = 15, b = 20;

    System.out.println("a y b antes de la llamada: " +
                       a + " " + b);

    ob.meth(a, b);

    System.out.println("a y b después de la llamada: " +
                       a + " " + b);
  }
}
```

A continuación se muestra la salida de este programa:

```
a y b antes de la llamada: 15 20
a y b después de la llamada: 15 20
```

Las operaciones que tienen lugar dentro de **meth( )** no tienen ningún efecto sobre los valores de **a** y **b** que se utilizan en la llamada; sus valores no se convierten en 30 y 10.

Cuando se pasa un objeto a un método, la situación cambia totalmente, ya que los objetos se pasan por referencia. Debe tenerse en cuenta que cuando se crea una variable cuyo tipo es una clase, lo que se está creando es una referencia a un objeto. Por lo tanto, cuando se pasa esta referencia a un método, el parámetro que la recibe se referirá al mismo objeto al que se refiere el argumento. Esto significa que los objetos se pasan a los métodos utilizando la llamada por referencia. Los cambios en el objeto dentro del método afectan al objeto utilizado como argumento. Por ejemplo, observe lo que sucede en el siguiente programa:

```
// Los objetos se pasan por referencia.

class Test {
  int a, b;

  Test(int i, int j) {
    a = i;
    b = j;
  }
  // paso del objeto
  void meth(Test o) {
    o.a *= 2;
    o.b /= 2;
  }
}

class LlamadaPorReferencia {
  public static void main(String args[]) {
    Test ob = new Test(15, 20);

    System.out.println("ob.a y ob.b antes de la llamada: " +
                       ob.a + " " + ob.b);

    ob.meth(ob) ;
```

```
    System.out.println("ob.a y ob.b después de la llamada: " +
                       ob. a + " " + ob.b);
  }
}
```

Este programa genera la siguiente salida:

```
ob.a y ob.b antes de la llamada: 15 20
ob.a y ob.b después de la llamada: 30 10
```

En este caso, las acciones efectuadas dentro de **meth( )** han afectado al objeto utilizado como argumento.

Una cuestión que se ha de tener en cuenta es que cuando se pasa una referencia a un método, la propia referencia se pasa por valor. Sin embargo, ya que el valor que se pasa se refiere a un objeto, la copia de ese valor sigue haciendo referencia al mismo objeto que su correspondiente argumento.

NOTA *Cuando se pasa un tipo simple a un método se hace por valor, mientras que los objetos se pasan por referencia.*

Devolución de objetos

Un método puede devolver cualquier tipo de dato, incluyendo los tipos definidos por el programador a través de clases. Por ejemplo, en el siguiente programa el método **increnDiez( )** devuelve un objeto en el que el valor de **a** es diez unidades mayor que en el objeto que llama al método.

```
// Devolución de un objeto.
class Test {
  int a;

  Test(int i) {
    a = i;
  }

  Test increnDiez() {
    Test temp = new Test(a+10);
    return temp;
  }
}

class RetOb {
  public static void main(String args[]) {
    Test obl = new Test(2);
    Test ob2;

    ob2 = obl. increnDiez ();
    System.out.println("obl.a: " + obl.a);
    System.out.println("ob2.a: " + ob2.a);

    ob2 = ob2.increnDiez();
```

```
    System.out.println("ob2.a después del segundo incremento: "
                       + ob2.a);
  }
}
```

La salida generada por este programa es la siguiente:

```
ob1.a: 2
ob2.a: 12
ob2.a después del segundo incremento: 22
```

Cada vez que se llama al método **increnDiez( ),** se crea un nuevo objeto, y se devuelve a la rutina llamante una referencia al mismo.

El programa anterior pone de manifiesto otro punto importante: considerando que los objetos se crean dinámicamente utilizando el operador **new**, no hay que preocuparse porque un objeto desaparezca cuando finaliza el método en que fue creado. El objeto seguirá existiendo siempre que en alguna parte del programa exista una referencia al mismo. Cuando no haya referencias será eliminado por el sistema de recolección de basura.

Recursividad

Java soporta la *recursividad*. La recursividad es el proceso mediante el que se define un ente en función de sí mismo. Por lo que se refiere al lenguaje de programación Java, la recursividad es el atributo que permite a un método llamarse a sí mismo. Un método con esta propiedad se denomina *recursivo*.

Un ejemplo clásico de recursividad es el cálculo del factorial de un número. El factorial de un número *N* es el producto de todos los números enteros entre 1 y *N*. Por ejemplo, factorial de 3 es igual a 1 × 2 × 3, o 6. A continuación se muestra cómo se puede calcular el factorial de un número mediante un método recursivo:

```
// Un ejemplo sencillo de recursividad.
class Factorial {
  // este es el método recursivo
  int fact(int n) {
    int result;

    if(n==1) return 1;
    result = fact(n-1) * n;
    return result;
  }
}

class Recursividad (
  public static void main(String args[]) {
    Factorial f = new Factorial();

    System.out.println("Factorial de 3 es " + f.fact(3));
    System.out.println("Factorial de 4 es " + f.fact(4));
    System.out.println("Factorial de 5 es " + f.fact(5));
  }
}
```

La salida generada por este programa es:

```
Factorial de 3 es 6
Factorial de 4 es 24
Factorial de 5 es 120
```

Si no está familiarizado con los métodos recursivos, entonces la operación del método **fact( )** puede resultarle un tanto compleja. Analicémosla con más detalle. Cuando se llama a **fact( )** con un argumento igual a 1, la función devuelve un 1; en caso contrario devuelve el producto de **fact(n-l)*n**. Para evaluar esta expresión se llama a **fact( )** con el argumento **n-1**. Este proceso se repite hasta que el valor de **n** es igual a 1 y las llamadas al método comienzan a devolver valores.

Para comprender mejor cómo funciona el método **fact( )**, veamos un ejemplo. Cuando se calcula el factorial de 3, la primera llamada a **fact( )** ocasiona una segunda llamada con un argumento igual a 2. Esta invocación hace que se vuelva a llamar a **fact( )** por tercera vez con un argumento igual a 1. Esta llamada devuelve el valor 1, que a continuación se multiplica por 2 (el valor de **n** en la segunda invocación). El resultado, que es 2, se devuelve a la primera invocación de **fact( )** y se multiplica por 3, el valor original de **n**. Esto da lugar a la respuesta, 6. Puede resultar interesante insertar una sentencia **println( )** en el cuerpo del método en **fact( )**, para mostrar en qué nivel se encuentra cada llamada y cuáles son las respuestas intermedias.

Cuando un método se llama a sí mismo, se almacenan nuevas variables locales y parámetros en la pila, y el código del método se ejecuta con estas nuevas variables desde el principio. Cuando una llamada recursiva devuelve los valores, las variables locales y parámetros antiguos se eliminan de la pila, y la ejecución continúa en el punto de llamada dentro del método. Se puede decir que los métodos recursivos se "expanden y contraen".

Las versiones recursivas de muchas rutinas pueden tener una ejecución ligeramente más lenta que su equivalente iterativa, debido a la sobrecarga ocasionada por las llamadas adicionales a métodos. Si se realizan muchas llamadas recursivas a un método, puede llenarse la pila del sistema, ya que el almacenamiento de los parámetros y variables locales se hace en la pila, y cada nueva llamada crea una copia nueva de estas variables. Si se llena la pila, el intérprete Java causará una excepción. Sin embargo, en general, no es necesario tener este hecho en cuenta, a menos que exista una rutina recursiva especialmente compleja.

La principal ventaja de los métodos recursivos es que se pueden utilizar para crear versiones más claras y sencillas de diferentes algoritmos que las correspondientes a métodos iterativos. Por ejemplo, es bastante difícil implementar el algoritmo QuickSort de forma iterativa. También algunos problemas relacionados con la inteligencia artificial, parecen conducir a soluciones recursivas.

Al escribir métodos recursivos, se debe tener una sentencia **if** en alguna parte del cuerpo del método para obligar al método a volver sin que la llamada recursiva sea ejecutada en algún momento. Si esto no se hace, una vez que se llama al método, éste ya no volverá. Éste es un error que se produce con frecuencia cuando se utiliza la recursividad. Una buena práctica es utilizar la sentencia **println( )** cuando se desarrolla el programa, de forma que se pueda ver qué es lo que está ocurriendo y abortar la ejecución si se comprueba que se ha cometido un error.

A continuación se presenta otro ejemplo de recursividad. El método recursivo **printArray( )** imprime los primeros **i** elementos del arreglo **values**.

```
// Otro ejemplo que utiliza la recursividad.

class RecTest {
  int values [];

  RecTest(int i) {
    values = new int[i];
  }

  // muestra los valores del arreglo -- recursivamente
  void printArray(int i) {
    if(i==0) return;
    else printArray(i-1);
    System.out.println("[" + (i-1) + "] " + values[i-1]);
  }
}

class Recursividad2 {
  public static void main(String args[]) {
    RecTest ob = new RecTest(10) ;
    int i;

    for(i=0; i<10; i++) ob.values[i] =i;

    ob.printArray(10);
  }
}
```

Este programa genera la siguiente salida:

```
[0] 0
[1] 1
[2] 2
[3] 3
[4] 4
[5] 5
[6] 6
[7] 7
[8] 8
[9] 9
```

Control de acceso

Como se ha comentado anteriormente, la encapsulación relaciona datos con el código que opera sobre los mismos. Pero, además, la encapsulación proporciona otro atributo importante: el *control de acceso.* A través de la encapsulación se puede controlar el acceso a los miembros de una clase desde las diferentes partes de un programa, y de esta forma impedir un mal uso de los mismos. Por ejemplo, si sólo se permite el acceso a los datos a través de métodos bien definidos, se impide una mala utilización de los mismos. Por lo tanto, cuando una clase está correctamente implementada, crea una "caja negra" que puede ser utilizada, pero cuyo funcionamiento interno no está abierto a la actuación exterior. Sin embargo, las clases

presentadas hasta el momento no cumplen por completo este objetivo. Considere, por ejemplo, la clase **Stack** que aparece al final del Capítulo 6. Si bien es cierto que los métodos **push( )** y **pop( )** proporcionan una interfaz controlada de la pila, esta interfaz no es la única, es decir, es posible que otra parte del programa acceda directamente a la pila evitando estos métodos. Evidentemente, esto puede provocar problemas si no se hace un uso correcto de la clase. En esta sección se introducirán los mecanismos mediante los cuales se puede controlar de manera precisa el acceso de los distintos miembros de la clase. Un especificador de acceso es responsable de determinar cómo se puede acceder a un miembro de la clase. El *especificador de acceso* se coloca en la declaración del miembro de la clase. Java facilita un amplio conjunto de especificadores de acceso. Algunos aspectos del control de acceso están directamente relacionados con la herencia o los paquetes (un *paquete* es esencialmente un grupo de clases). Estas partes de mecanismo de control de acceso de Java se discutirán más adelante, por lo pronto comencemos examinando el control de acceso tal y como se aplica a una sola clase. Una vez que se entiendan claramente los fundamentos del control de acceso, el resto resultará sencillo.

Los especificadores de acceso de Java son **public**, **private** y **protected**. Java también define un nivel de acceso por omisión. El especificador **protected** se aplica solamente en el caso de que se trabaje con herencia. Los demás especificadores de acceso se describen a continuación.

Comencemos definiendo **public** y **private**. Cuando se aplica a un miembro de una clase el especificador **public**, entonces se puede acceder a ese miembro por cualquier código del programa. Cuando se especifica un miembro de una clase como **private**, entonces sólo se puede acceder a ese miembro desde otros miembros de su clase. Ahora resulta sencillo entender por qué **main( )** siempre ha sido precedido por el especificador **public**, permitiendo el acceso al mismo desde fuera del programa, es decir, desde el intérprete Java. Cuando no se utiliza un especificador de acceso, entonces, por omisión, se considera que el miembro de la clase es público dentro de su propio paquete, pero no se puede acceder al mismo desde fuera de su paquete. (En el próximo capítulo se analizarán los paquetes).

En las clases desarrolladas hasta el momento se ha utilizado el modo de acceso por omisión, que es esencialmente el público. Sin embargo, éste no será el caso general en la vida real. Normalmente, se deseará un acceso restringido a los datos de una clase, permitiendo el acceso sólo a través de los métodos. También, en otras ocasiones, será deseable definir métodos privados para una clase.

Un especificador de acceso precede al resto de especificaciones de tipo de un miembro, es decir, la sentencia de declaración de cualquier miembro de la clase va precedida por su especificador de acceso.

```
public int i;
private double j,

private int miMetodo(int a, char b) { // …
```

Para comprender mejor el significado del acceso público y privado a miembros de una clase, veamos el siguiente programa de ejemplo:

```
/* Este programa muestra la diferencia entre
   acceso público y privado.
*/
class Test {
```

```
  int a; // acceso por omisión
  public int b; // acceso público
  private int c; // acceso privado

  // métodos para acceder a c
  void setc(int i) { // establece el valor de c
    c = i;
  }
  int getc() { // se obtiene el valor de c
    return c;
  }
}

class TestAcceso {
  public static void main(String args[]) {
    Test ob = new Test();

    // Esto es correcto y se puede acceder directamente a a y b
    ob.a = 10;
    ob.b = 20;

    // Esto no es correcto y causará un error
// ob.c = 100; // error

    // Se debe acceder a c a través de sus métodos
    ob.setc(100); // correcto
    System.out.println ("a, b y c: " + ob.a + " " + ob.b + " " +
                        ob.getc());
  }
}
```

Dentro de la clase **Test**, **a** utiliza el acceso por omisión, que en este ejemplo es lo mismo que especificar como **public**. La variable **b** se especifica explícitamente como public, mientras que el acceso a **c** es privado. Esto significa que no se puede acceder a **c** desde un código que esté fuera de su clase. Así que no se puede acceder directamente a **c** desde la clase **TestAcceso**, sino que se debe hacer a través de los métodos públicos: **setc( )** y **getc( )**. Si se elimina el símbolo de comentario del comienzo de la siguiente línea,

```
// ob.c = 100; // error
```

Entonces el programa no compila, debido a la violación del acceso.

Para comprobar cómo se puede aplicar el control de acceso a un ejemplo más práctico, consideremos la siguiente versión mejorada de la clase **Stack** que vimos al final del Capítulo 6.

```
// Esta clase define una pila de enteros que puede contener 10 valores.
class Stack {
  /* Ahora, las variables stck y tos son privadas. Que
    no pueden ser modificadas de forma accidental o intencionada,
    con lo que evitamos resultados perjudiciales para la pila.
*/
private int stck[] = new int[10];
private int tos;
```

```
  // Inicialización de la posición superior de la pila
  Stack () {
    tos = -1;
  }

  // Se introduce un dato en la pila
  void push (int item) {
    if(tos==9)
      System.out.println("La pila está llena.");
    else
      stck[++tos] = item;
  }
  // Se retira un dato de la pila
  int pop() {
    if(tos < 0) (
      System.out.println("La pila está agotada.");
      return 0;
    }
    else
      return stck[tos-];
  }
}
```

Ahora, tanto **stck**, que almacena la pila, como **tos**, que es el índice de la parte superior de la pila, se especifican como **private**. Esto significa que no pueden ser modificados o que no se puede acceder a ellos más que mediante **push( )** y **pop( )**. Especificando **tos** como privado, se impide, por ejemplo, que otras partes del programa le den un valor más allá del final del arreglo **stck**.

El siguiente programa muestra la mejora efectuada en nuestra clase **Stack**. Intente eliminar los comentarios de las líneas marcadas para comprobar que no se puede acceder a los miembros **stck** y **tos**.

```
class TestStack {
  public static void main{String args[]) {
    Stack miPila1 = new Stack();
    Stack mipila2 = new Stack();

    // se introducen algunos números en la pila
    for(int i=0; i<l0; i++) miPila1.push{i);
    for(int i=l0; i<20; i++) miPila2.push{i);

    // se recuperan números de la pila
    System.out.println("Contenido de miPila1:");
    for(int i=0; i<l0; i++)
      System.out.println(miPila1.pop());

    System.out.println("Contenido de miPila2:"):
    for(int i=0; i<l0;i++)
      System.out.println(miPila2.pop()):

    // estas sentencias no son válidas
    // miPilal.tos = -2:
    // miPila2.stck[3] = 100:
  }
}
```

Aunque los métodos proporcionan normalmente acceso a los datos definidos por una clase, esto no tiene por qué ser siempre así. Es perfectamente válido que una variable de instancia sea pública cuando existan razones para ello. Por ejemplo, la mayor parte de las clases que aparecen en este libro se han creado sin tener en cuenta el control de acceso a variables de instancia para simplificar los ejemplos. Sin embargo, en la mayor parte de las aplicaciones prácticas será necesario que las operaciones sobre los datos se realicen solamente a través de los métodos. En el próximo capítulo se vuelve al tema del control de acceso. Este tema es de particular importancia cuando se consideran cuestiones de herencia.

static

En ocasiones puede ser necesario definir un miembro de clase que será utilizado independientemente de cualquier objeto de esa clase. Normalmente se accede a un miembro de clase asociado con un objeto de su clase. Sin embargo, es posible crear un miembro que pueda ser utilizado por sí mismo, sin referencia a una instancia específica. Para crear un miembro de este tipo es necesario que su declaración vaya precedida de la palabra clave **static**. Cuando se declara a un miembro como **static**, se puede acceder al mismo antes de que cualquier objeto de su clase sea creado, y sin referencia a ningún objeto. Se pueden declarar tanto a los métodos como a variables como **static**. El ejemplo más habitual de un miembro **static** es **main( ). main( )** se declara como **static**, ya que debe ser llamado antes de que exista cualquier objeto.

Las variables de instancia declaradas como **static** son, esencialmente, variables globales. Cuando se declaran los objetos de su clase no se hace ninguna copia de las variables declaradas **static**. En su lugar, todas las instancias de la clase comparten la misma variable **static**.

Los métodos declarados como **static** tienen varias restricciones:

- Sólo pueden llamar a otros métodos declarados como **static**.
- Sólo deben acceder a datos declarados como **static**.
- No pueden referirse a **this** o **super** de ninguna manera. La palabra clave **super** está relacionada con la herencia y se describe en el próximo capítulo.

Cuando sea necesario realizar cálculos para inicializar las variables de tipo **static**, se puede declarar como **static** un bloque que se ejecuta una sola vez, cuando se carga la clase por primera vez. El siguiente ejemplo muestra una clase que tiene un método **static**, algunas variables **static** y un bloque de inicialización **static**:

```
// Ejemplo de variables, métodos y bloques static.
class UsoStatic {
  static int a = 3;
  static int b;

  static void metodo(int x) {
    System.out.println("x = " + x);
    System.out.println("a = " + a);
    System.out.println("b = " + b);
  }

  static {
    System.out.println("Inicialización del bloque static.");
```

```
    b = a * 4;
  }

  public static void main(String args[]) {
    metodo (42) ;
  }
}
```

Tan pronto como la clase **UsoStatic** se carga, se ejecutan todas las sentencias **static**. En primer lugar, se asigna a la variable **a** el valor **3**; después se ejecuta el bloque **static** (se imprime un mensaje), y, finalmente, se inicializa la variable **b** haciéndola igual a **a * 4** ó **12**. A continuación se llama al método **main( )**, que llama a **metodo( )**, pasando el valor **42** a **x**. Las tres sentencias **println( )** se refieren a las dos variables **static a** y **b**, así como a la variable local **x**.

Ésta es la salida del programa:

```
Inicialización del bloque static.
x = 42
a = 3
b = 12
```

Fuera de la clase en la que se han definido, los métodos y las variables **static** se pueden utilizar independientemente de cualquier objeto. Para hacerlo sólo es necesario especificar el nombre de su clase seguido por el operador punto. Por ejemplo, si se desea llamar a un método **static** desde fuera de su clase, se puede hacer utilizando la siguiente forma general:

nombre_de_clase.metodo( )

Donde, *nombre_de_clase* es el nombre de la clase en la que se declaró el método **static**. Como puede observarse, este formato es semejante al utilizado para llamar a métodos no **estáticos** por medio de variables que se refieren a objetos. Se puede acceder a una variable estática de la misma forma, mediante el uso del operador punto y el nombre de la clase. De esta forma Java implementa una versión controlada de las funciones y variables globales.

El siguiente es un ejemplo de lo anterior, en donde dentro de **main( )**, se accede al método static **llamame( )** y a la variable **estática b** mediante el nombre de su clase.

```
class StaticDemo {
  static int a = 42;
  static int b = 99;
  static void llamame() {
    System.out.println("a = " + a);
  }
}

class StaticporNombre {
  public static void main{String args[]) {
    StaticDemo.llamame();
    System.out.println("b = " + StaticDemo.b);
  }
}
```

La salida de este programa es:

```
a = 42
b = 99
```

final

Declarando una variable como **final** se impide que su contenido sea modificado. Esto significa que una variable declarada como **final** debe ser inicializada cuando es definida. Por ejemplo:

```
final int NUEVO_ARCHIVO = 1;
final int ABRIR_ARCHIVO = 2;
final int GUARDAR_ARCHIVO = 3;
final int GUARDAR_ARCHIVO_COMO = 4;
final int ABANDONAR_ARCHIVO = 5;
```

Ahora se pueden utilizar **ABRIR_ARCHIVO** y el resto de las variables en el programa, como si fueran constantes, esto es, sin temor a que sus valores sean modificados.

Una convención muy utilizada en programación es la de utilizar especificadores en mayúsculas para las variables declaradas con el modificador **final**. Las variables declaradas como **final** no ocupan memoria en cada instancia, es decir, una variable **final** es básicamente una constante.

La palabra clave **final** también se puede aplicar a los métodos, pero su significado en este caso es sustancialmente distinto que cuando es aplicado a variables, este segundo uso de **final** se describe en el siguiente capítulo, cuando trataremos todos los temas relacionados con el concepto de herencia.

Más información sobre arreglos

Los arreglos fueron descritos en este texto antes de abordar el tema de las clases. Una vez que se han presentado las clases, ya podemos citar una característica importante de los arreglos; los arreglos se implementan como objetos. Esto permite aprovechar un atributo especial de los arreglos de datos: el tamaño de una matriz de datos, esto es, el número de elementos que puede contener, se encuentra en su variable de instancia **length**. Todos los arreglos tienen esta variable la cual contiene el tamaño del arreglo. El programa a continuación muestra esta propiedad:

```
// Este programa muestra como conocer el tamaño
   de un arreglo utilizando la variable length
class Length {
  public static void main(String args[]) {
    int a1[] = new int[10];
    int a2[] = {3, 5, 7, 1, 8, 99, 44, -10};
    int a3 [] = {4, 3, 2, 1};
    System.out.println("La longitud de a1 es " + a1.length);
    System.out.println("La longitud de a2 es " + a2.length);
    System.out.println("La longitud de a3 es " + a3.length);
  }
}
```

Este programa presenta la siguiente salida:

```
La longitud de a1 es 10
La longitud de a2 es 8
La longitud de a3 es 4
```

El programa despliega el tamaño de cada arreglo. Conviene tener presente que el valor de **length** no tiene nada que ver con el número de elementos que realmente están en uso. Solamente refleja el número de elementos que el arreglo puede contener.

La variable **length** puede ser muy útil en muchas situaciones. Como ejemplo, a continuación se presenta una versión mejorada de la clase **Stack**. Las versiones anteriores de esta clase siempre creaban una pila de 10 elementos. La siguiente versión nos permite crear pilas de cualquier tamaño. El valor de **stck**.**length** se utiliza para impedir el desbordamiento de la pila.

```
// Versión mejorada de la clase Stack que valida la longitud del arreglo
class Stack {
  private int stck[];
  private int tos;

  // asignación de memoria e inicialización de la pila
  Stack(int tamaño) {
    stck = new int[tamaño];
    tos = -1;
  }

  // Introduce un dato en la pila
  void push(int item) {
    if(tos==stck.length-1) // aquí se utiliza la variable length del arreglo
      System.out.println("La pila está llena");
    else
      stck[++tos] = item;
  }

  // Retira un dato de la pila
  int pop () {
    if (tos < 0) (
      System.out.println("La pila está vacía.") ;
      return 0;
    }
    else
      return stck[tos-];
  }
}

class TestStack2 (
    public static void main(String args[]) {
    Stack miPila1 = new Stack(5);
    Stack miPi1a2 = new Stack(8);
    // Introduce algunos números en la pila
    for(int i=0; i<5; i++) miPila1.push(i);
    for(int i=0; i<8; i++) miPila2.push(i);

    // Retira esos números de la pila
    System.out.println("Contenido de miPi1a1:");
```

```
    for(int i=0; i<5; i++)
      System.out.println(miPila1.pop());

    System.out.println("Contenido de miPila2:");
    for(int i=0; i<8; i++)
      System.out.println(miPila2.pop());
  }
}
```

Este programa crea dos pilas, una con cinco elementos y la otra con ocho. El hecho de que los arreglos contengan información sobre su propia longitud facilita la creación de pilas de cualquier tamaño.

Introducción a clases anidadas y clases interiores

Es posible definir una clase dentro de otra clase; tales clases reciben el nombre de *clases anidadas*. El campo de acción de una clase anidada se limita a la clase que la contiene. Es decir, si la clase B se define dentro de la clase A, entonces B es conocida dentro de A, pero no fuera de A. Una clase anidada tiene acceso a los miembros, incluyendo miembros privados, de la clase en la que está anidada. Sin embargo, la clase que la contiene no tiene acceso a los miembros de la clase anidada. Una clase anidada se considera un miembro de la clase que la contiene. También es posible declarar clases anidadas locales a un bloque en particular.

Existen dos tipos de clases anidadas: *estática* y *no-estática*. Una clase anidada del tipo estático es una clase a la que se aplica el modificador **static**. Como es estática, debe acceder a los miembros de la clase que la contiene por medio de un objeto, es decir, no puede hacer referencia directamente a los mismos. A causa de esta restricción las clases anidadas del tipo estático se usan muy poco.

El tipo más importante de clases anidadas es la clase *interior*. Una clase interior es una clase anidada no estática. Esta clase tiene acceso a todos los métodos y variables de la clase que la contiene y puede referirse a los mismos directamente, de la misma forma que otros miembros no estáticos de la clase exterior.

El siguiente programa describe cómo se define y usa una clase interior. La clase denominada **Exterior** tiene una variable de instancia que se llama **x_exterior**, un método de instancia denominado **test( )**, y define una clase interior que se llama **Interior**.

```
// Ejemplo de una clase interior.
class Exterior {
  int x_exterior = 100;

  void test() {
    Interior interior = new Interior();
    interior.display();
  }

  // ésta es una clase interior
  class Interior {
    void display() {
      System.out.println("imprime: x_exterior = " + x_exterior);
    }
  }
}
```

```
class ClaseInteriorDemo {
  public static void main(String args[]) {
    Exterior exterior = new Exterior();
    exterior.test () ;
  }
}
```

La salida de esta aplicación es la que se muestra a continuación:

```
imprime: x_exterior = 100
```

En el programa se define una clase interior denominada **Interior** dentro del campo de acción de la clase **Exterior**. Por lo tanto, cualquier código que esté dentro de la clase **Interior** puede acceder directamente a la variable **x_exterior**. Dentro de la clase Interior se define un método de instancia denominado **display( ).** Este método muestra **x_exterior** en el dispositivo de salida estándar. El método **main( )** de la clase **ClaseInteriorDemo** crea una instancia de clase **Exterior** y llama a su método **test( ).** Ese método crea una instancia de clase **Interior** y llama al método **display( ).**

Es importante tener en cuenta que la clase **Interior** solamente es conocida dentro del campo de acción de la clase **Exterior**. El compilador Java genera un mensaje de error si cualquier código que esté fuera de la clase **Exterior** intenta instanciar a la clase **Interior**. En general, una clase anidada no es diferente a cualquier otro elemento del programa, y es conocida sólo dentro del campo de acción de la clase que la contiene. Sin embargo, es posible crear una instancia de **Interior** fuera del ámbito de la clase **Exterior** utilizando el identificador completo **Exterior. Interior**.

Tal y como se ha explicado, una clase interior tiene acceso a todos los miembros de la clase que la contiene, pero no al revés. Los miembros de una clase interior sólo se conocen dentro del campo de acción de la clase interior y no pueden ser utilizados por la clase exterior. Por ejemplo,

```
// Este programa no se compilará.
class Exterior {
  int x_exterior = 100;

  void test() {
    Interior interior = new Interior();
    interior.display();
  }

  // Esta es una clase interior
  class Interior {
    int y = 10; // y es local para Interior
    void display() {
      System.out.println("presenta: x_exterior = " + x_exterior);
    }
  }

  void muestray () {
    System.out.println(y); // error, y es desconocido aquí!
  }
}

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

```
    Exterior exterior = new Exterior();
    exterior.test ();
  }
}
```

Aquí, **y** se declara como una variable de instancia de la clase **Interior**, es decir, será desconocida fuera de esa clase y no podrá ser utilizada por **muestray( ).**

Aunque nos hemos centrado en las clases anidadas declaradas dentro del campo de acción de una clase exterior, también es posible definir clases interiores dentro del campo de acción de un bloque. Por ejemplo, se puede definir una clase anidada dentro del bloque definido por un método, o incluso dentro del cuerpo de un bucle **for**, tal y como muestra el siguiente programa.

```
// Definición de una clase interior dentro de un bucle for
class Exterior {
  int x_exterior = 100;

  void test() {
    for(int i=0; i<10; i++) {
      class Interior {
        void display() {
          System.out.println("muestra: x_exterior = " + x_exterior);
        }
      }
      Interior interior = new Interior();
      interior.display();
    }
  }
}

class ClaseInteriorDemo {
  public static void main(String args[]) {
    Exterior exterior = new Exterior();
    exterior.test ( ) ;
  }
}
```

La salida de esta versión del programa es la siguiente:

```
muestra: x_exterior = 100
muestra: x_exterior = 100
muestra: x_exterior = 100
muestra: x_exterior = 100
muestra: x_exterior = 100
muestra: x_exterior = 100
muestra: x_exterior = 100
muestra: x_exterior = 100
muestra: x_exterior = 100
muestra: x_exterior = 100
```

Aunque las clases anidadas no se usan habitualmente, son particularmente útiles en la gestión de eventos. Regresaremos al tópico de las clases anidadas en el Capítulo 22, donde se verá cómo se pueden utilizar las clases interiores para simplificar el código necesario para gestionar cierto tipo de eventos. También se tratarán las *clases interiores anónimas,* que son clases interiores sin nombre.

Una anotación final: Las clases anidadas no se permitían dentro de las especificaciones de Java 1.0. Fueron incorporadas por lava 1.1.

La clase String

Aunque la clase String se estudiará con detalle en la Parte II de este libro, es preciso hacer una introducción de la misma en este momento, ya que utilizaremos objetos **String** en algunos de los programas de ejemplo que aparecen más adelante en la Parte 1 de este libro. La clase **String** es probablemente la más utilizada de la biblioteca de clases de Java. El motivo de ello es que las cadenas de caracteres son una parte muy importante de la programación.

La primera cuestión que se ha de tener en cuenta en relación con las cadenas de caracteres es que son realmente un objeto del tipo **String**, incluso en el caso de constantes. Por ejemplo, en la sentencia

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

la cadena "Esto también es una cadena de caracteres"es una constante **String**.

La segunda cuestión a considerar es que los objetos del tipo **String** son inmutables; es decir, una vez que se crea un objeto del tipo **String** su contenido no se puede modificar. Esto, que puede parecer una restricción importante, no lo es por dos razones:

- Si se necesita cambiar una cadena, siempre se puede crear una nueva que contenga las modificaciones.
- Java define una clase equivalente de **String**, denominada **StringBuffer**, que permite modificar las cadenas de forma que las manipulaciones habituales de cadenas sean posibles en Java. La clase **StringBuffer** se describe en la Parte II de este libro.

Las cadenas de caracteres se pueden construir de muchas formas. La más sencilla es utilizar una sentencia como ésta:

```
String miCadena="esto es una prueba";
```

Una vez que se crea un objeto **String**, se puede utilizar en cualquier parte en la que una cadena esté permitida. Por ejemplo, esta sentencia imprime el valor de la variable **miCadena**:

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

Java define un operador para objetos **String**: +. Este operador se utiliza para concatenar dos cadenas. Por ejemplo, esta sentencia

```
String miCadena = "Me" + " gusta " + "Java.";
```

da como resultado que **miCadena** contenga "Me gusta Java." Como ejemplo de los conceptos anteriores se presenta el siguiente programa:

```
// Ejemplo de cadenas de caracteres.
class StringDemo {
  public static void main(String args[]) {
    String strObl = "Primera cadena";
    String strOb2 = "Segunda cadena";
    String strOb3 = strObl + " y " + strOb2;
```

```
    System.out.println(strObl);
    System.out.println(strOb2) ;
    System.out.println(strOb3);
  }
}
```

A continuación se muestra la salida que se obtiene con este programa:

```
Primera cadena
Segunda cadena
Primera cadena y segunda cadena
```

La clase **String** contiene varios métodos que pueden ser utilizados. Entre los más importantes se encuentran los siguientes. Se puede comprobar la igualdad de dos cadenas mediante **equals( )**. Se puede obtener la longitud de una cadena llamando al método **length( ).** Se puede obtener el carácter que ocupa una posición determinada dentro de una cadena llamando al método **charAt( ).** La forma general de estos tres métodos es la que aparece a continuación:

boolean equals(*objeto* String)
int length( )
char charAt(int *índice*)

Aquí se presenta un ejemplo de estos tres métodos en el siguiente programa:

```
// Ejemplo de algunos métodos de String
class StringDemo2 {
  public static void main(String args[]) {
    String strObl = "Primera cadena";
    String strOb2 = "Segunda cadena";
    String strOb3 = strObl;

    System.out.println("Longitud de strObl:" +
                       strObl.length());

    System.out.println("El carácter en la posición 3 de strObl: " +
                       strObl.charAt(3));

    if(strObl.equals(strOb2))
      System.out.println("strObl == strOb2");
    else
      System.out.println("strObl != strOb2");

    if(strObl.equals(strOb3))
      System.out.println("strObl == strOb3");
    else
      System.out.println("strObl != strOb3");
  }
}
```

Este programa genera la siguiente salida:

```
Longitud de strOb1: 14
El carácter en la posición 3 de strOb1: m
strOb1 != strOb2
strOb1 == strOb3
```

Naturalmente existen arreglos de cadenas de caracteres, del mismo modo que existen arreglos de cualquier otro tipo de objetos. Por ejemplo:

```
// Ejemplo de arreglos de cadenas de caracteres.
class StringDemo3 {
  public static void main(String args[]) {
    String str [] = { "uno", "dos", "tres" };

    for(int i=0; i < str.length; i++)
      System.out.println("str[" + i + "]: " + str[i] ) ;
  }
}
```

La salida que se obtiene es la siguiente:

```
str [0]: uno
str [1]: dos
str [2]: tres
```

Como se verá en el siguiente apartado, los arreglos de cadenas de caracteres desempeñan un papel importante en muchos programas de Java.

Argumentos en la línea de órdenes

En ocasiones, es necesario pasar información a un programa que se está ejecutando. Esto se lleva a cabo pasando *argumentos desde la línea de órdenes* a **main( ).** Un argumento de la línea de órdenes es la información que va inmediatamente después del nombre del programa en la línea de órdenes cuando es ejecutado. Resulta sencillo acceder a los argumentos de la línea de órdenes dentro de un programa Java, ya que los mismos se almacenan como cadenas de caracteres en el arreglo de tipo **String** que se pasa a **main( )** en el parámetro **args**. El primer argumento se almacena en **args[0]**, el segundo en **args[1]**, y así sucesivamente. Por ejemplo, el siguiente programa presenta todos los argumentos que recibe la línea de órdenes.

```
// Este ejemplo despliega los argumentos que recibe desde la línea
   de órdenes.
class LineaDeOrdenes {
  public static void main(String args[]) {
    for(int i=0; i<args.length; i++)
      System.out.println("args[" + i + "]: " +
                         args[i]);
  }
}
```

Intente ejecutar este programa tal y como se muestra aquí:

```
java LineaDeOrdenes esto es una prueba 100 -1
```

Al hacerlo se obtiene la siguiente salida:

```
args[0]: esto
args[1]: es
args[2]: una
args[3]: prueba
args[4]: 100
args[5]: -1
```

NOTA *Todos los argumentos de la línea de órdenes se pasan como cadenas de caracteres. Se debe convertir los valores numéricos a su formato interno manualmente, tal y como se explica en el Capítulo 16.*

Argumentos de tamaño variable

Comenzando con JDK5, Java ha incluido una característica que simplifica la creación de métodos que necesitan tomar un número variable de argumentos. Esta característica es llamada *varargs* y es la abreviatura en inglés de *argumentos de tamaño variable*. Un método que toma un número variable de argumentos está llamando un *método de grado variable* o simplemente un *método varargs.*

Las situaciones que requieren que un número variable de argumentos sea pasado a un método no son usuales. Por ejemplo, un método que abre una conexión de Internet podría tomar un nombre de usuario, una contraseña, un nombre de archivo, un protocolo y así sucesivamente, pero habrá casos en los que se desee tomar valores por omisión si alguna de esta información no es proporcionada. En esta situación, sería conveniente pasar solo los argumentos que no cuentan con un valor por omisión. Otro ejemplo es el método **printf()** que es parte de las biblioteca de E/S de Java. Como se verá en el Capítulo 19, dicho método toma un número variable de argumentos, a los cuales da formato y muestra en pantalla.

Antes de JDK 5, los argumentos de tamaño variable, podían ser gestionados de dos formas, ninguna de las cuales era particularmente agradable. La primera, si el máximo número de argumentos era pequeño y conocido, entonces era viable crear varias versiones del método utilizando sobrecarga, una para cada forma en que el método podría ser llamado. Aunque esto funciona y es apropiado para algunos casos, no es aplicable para cualquier situación.

En el caso en donde el número máximo de argumentos potenciales era largo, o desconocido, una segunda forma era utilizada. En esta segunda forma el número de argumentos era colocado en un arreglo, y luego el arreglo era pasado al método. Esta forma de solución se ilustrada en el siguiente programa.

```
// Uso de un arreglo para pasar un numero variable de argumentos
// a un método. Éste es el estilo antiguo de resolver el problema
// de argumentos de tamaño variable
class PassArray {
  static void vaPrueba(int v[]) {
    System.out.print("Número de argumentos: "+ v.length  +
                     "Contenido: ");

    for (int x : v)
      System.out.print(x + " ");
      System.out.println();
    }

  public static void main(String args[])
  {
    // Observe como un arreglo de ser creado para
    // almacenar los argumentos
    int n1 [] = { 10 };
    int n2 [] = { 1, 2, 3 };
    int n3 [] = { };
```

```
    vaPrueba (n1); // 1 argumento
    vaPrueba (n2); // 3 argumentos
    vaPrueba (n3); // sin argumentos
  }
}
```

La salida de este programa se muestra a continuación

```
Número de argumentos: 1 Contenido: 10
Número de argumentos: 3 Contenido: 1 2 3
Número de argumentos: 0 Contenido:
```

En el programa, el método **vaPrueba( )** está pasando argumentos a través del arreglo **v.** Este viejo estilo de trabajar con argumentos de tamaño variable permite al método **vaPrueba( )** recibir un número arbitrario de argumentos. Sin embargo, requiere que esos argumentos sean manualmente empacados dentro de un arreglo antes de llamar al método **vaPrueba( ).** No sólo es tedioso construir un arreglo cada vez que **vaPrueba( )** es llamado, es también una forma de trabajo potencialmente propensa a errores. La característica varargs ofrece una mejor y simple opción.

Un argumento de tamaño variable se especifica por tres puntos (…). Por ejemplo, a continuación se muestra como **vaPrueba( )** se escribiría utilizando vararg:

```
static void vaPrueba(int … v) {
```

Esta sintaxis le dice al compilador que **vaPrueba( )** puede ser llamado con cero o más argumentos. Como un resultado, **v** es implícitamente declarada como un arreglo de tipo **int[].** Así, dentro de **vaPrueba( ), v** es accedido usando la sintaxis normal de arreglos. El programa anterior quedaría de la siguiente forma utilizando vararg:

```
// Ejemplo de argumentos de tamaño variable
class VarArgs {

  // vaPrueba() ahora utilizando vararg
  static void vaPrueba(int ... v) {
    System.out.print("Número de argumentos: " + v.length +
                     " Contenido: ");

    for (int x : v)
      System.out.print(x + " ");

    System.out.println() ;
  }

public static void main(String args[])
{

    // Observe como vaPrueba() puede ser llamado con
    // un número variable de argumentos
    vaTest(l0); // 1 argumento
    vaTest(l, 2, 3); // 3 argumentos
    vaTest(); // sin argumentos
  }
}
```

La salida de este programa es igual a la versión original.

Hay dos puntos importantes a considerar acerca de este programa. En primer lugar, como se explicó, dentro de **vaPrueba( )**, **v** es utilizado como un arreglo. Esto es porque v es un arreglo. La sintaxis de (…) simplemente le dice al compilador que un número variable de argumentos serán utilizados, y que esos argumentos serán guardados en el arreglo llamado **v**. En segundo lugar, en el método **main( )**, **vaPrueba( )** es llamado con diferente número de argumentos, incluyendo una llamada sin ningún argumento. Estos argumentos son automáticamente puestos en un arreglo y pasados a **v**. En el caso de no argumentos, la longitud del arreglo es cero.

Un método puede tener parámetros "normales" junto con parámetros de tamaño variable. Sin embargo, el parámetro de longitud variable debe ser el último parámetro declarado por el método. Por ejemplo, la declaración de este método es perfectamente aceptable:

```
int Hazlo(int a, int b, double c, int … vals) {
```

En este caso, los primeros tres argumentos usados en una llamada a **Hazlo( )** corresponderán a los primeros tres parámetros. Y cualquier argumento restante se asumirá que pertenece a **vals**.

Recuerde, que el parámetro varargs debe ser el último. Por ejemplo, la siguiente declaración es incorrecta:

```
int Hazlo(int a, int b, double c, int … vals, boolean stopFlag) { // error
```

El intento de declarar un parámetro regular después del argumento varargs es ilegal.

Una restricción más a considerar es que debe haber solamente un parámetro varargs. Por ejemplo, esta declaración también es inválida:

```
int Hazlo(int a, int b, double c, int … vals, double … masvals) { // error
```

El intento de declarar un segundo argumento varargs es ilegal.

A continuación un nueva versión del método **vaPrueba( )** que utiliza un argumento regular y un argumento de tamaño variable

```
// Uso de varargs y argumentos estándares en el mismo metodo
class VarArgs2 {

  // msg es un parámetro normal y v es un
  // parámetro varargs
  static void vaPrueba(String msg, int ... v) {
    System.out.print(msg + v.length +
                       " Contenido: ");

    for(int x : v)
      System.out.print(x+ " ");

    System.out.println();
  }

  public static void main(String args[])
  {

    vaPrueba ("Un varargs: ", 10);
    vaPrueba ("Tres varargs: ", 1, 2, 3);
    vaPrueba ("Sin varargs: ");
  }
}
```

La salida del programa se muestra a continuación:

```
Un vararg: 1 Contenido: 10
Tres varargs: 3 Contenido: 1 2 3
Sin varargs: 0 Contenido:
```

Sobrecarga de métodos con argumentos de tamaño variable

Se puede sobrecargar un método que toma argumentos de tamaño variable. Por ejemplo, el siguiente programa sobrecarga el método **vaPrueba()** tres veces:

```
// varargs y sobrecarga
class VarArgs3 {

  static void vaPrueba(int ... v) {
    System.out.print ("vaPrueba (int ...): " +
                      "Número de argumentos: " + v.length +
                      " Contenido: ");

    for (int x : v)
      System.out.print(x + " ");

    System.out.println();
  }

  static void vaPrueba(boolean ... v) {
    System.out.print ("vaPrueba (boolean ...) " +
                      "Número de argumentos: " + v.length +
                      " Contenido: ");

    for(boolean x : v)
      System.out.print(x + " ");

    System.out.println();
  }

  static void vaPrueba(String msg, int ... v) {
    System.out.print ("vaPrueba (String, int …): "+
                      msg + v.length +
                      " Contenido: ");

    for(boolean x : v)
      System.out.print(x + " ");

    System.out.println() ;
  }

  public static void main(String args[])
  {
    vaPrueba(l, 2, 3);
    vaPrueba("Probando: ", 10, 20);
    vaPrueba(true, false, false);
  }
}
```

A continuación se muestra la salida del programa:

```
vaPrueba(int ...): Número de argumentos: 3 Contenido:  1 2 3
```

```
vaPrueba(String, int ...): Probando: 2 Contenido:: 10 20
vaPrueba(boolean ...) Número de argumentos: 3 Contenido:  true false false
```

Este programa ilustra ambas formas en que el método varargs puede ser sobrecargado. En primer lugar, el tipo del parámetro vararg puede ser diferente. En este caso para **vaPrueba(int …)** y **vaPrueba(bolean…).** Recuerde que los (…) causan que el parámetro sea tratado como un arreglo del tipo especificado. Por consiguiente, así como es posible sobrecargar métodos usando diferentes tipos de arreglos, también es posible sobrecargar métodos varargs usando diferentes tipos de varargs. En este caso, Java usa las diferencias de tipos para determinar cuál de los métodos sobrecargados llamar.

La segunda forma de sobrecargar los métodos varargs es agregando parámetros normales. Esto es, lo que se hizo con **vaPrueba(String, int …).** En este caso, Java utiliza tanto el número de argumentos como el tipo de los argumentos para determinar cuál método llamar.

NOTA *Un método varargs puede también ser sobrecargado por un método sin varargs. Por ejemplo,* ***vaPrueba(int x)*** *es una sobrecarga valida del método* ***vaPrueba( )*** *en el programa anterior. Esta versión es invocada sólo cuando un único argumento* ***int*** *está presente. Cuando dos o más argumentos* ***int*** *son pasados, la versión del método con varargs,* ***vaPrueba(int …v)****, es utilizada.*

Argumentos de tamaño variable y ambigüedad

Pueden ocurrir errores inesperados cuando se sobrecarga un método que toma argumentos de tamaño variable. Estos errores involucran ambigüedad porque es posible crear una llamada ambigua para un método varargs sobrecargado. Por ejemplo, considere el siguiente programa.

```
// varargs, sobrecarga, y ambigüedad
//
// Este programa contiene un error y
// no compilara
class VarArgs4 {

  static void vaPrueba(int ... v) {
    System.out.print("vaPrueba(int ...): " +
                     "Número de argumentos: " + v.length +
                     " Contenido: ");

    for (int x : v)
      System.out.print(x + " ");

    System.out.println() ;
  }

  static void vaPrueba(boolean ... v) {
    System.out.print("vaPrueba(boolean ...) " +
                     "Número de argumentos: " + v.length +
                     " Contenido: ");

    for (boolean x : v)
      System.out.print(x + " ");

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

```
  public static void main(String args[])
  {
    vaPrueba(1, 2, 3); // correcto
    vaPrueba(true, false, false); // correcto

    vaPrueba(); // esto genera un error de ambigüedad
  }
}
```

En este programa, la sobrecarga de **vaPrueba( )** es perfectamente correcta. Sin embargo, este programa no compilará debido a la siguiente llamada:

```
vaPrueba(); // esto genera un error de ambigüedad
```

Debido a que el parámetro vararg puede estar vacío, esta llamada podría ser trasladada dentro de una llamada a **vaPrueba(int …)** o **vaPrueba(boolean …).** Ambos son igualmente válidos. Así, la llamada es esencialmente ambigua.

A continuación se presenta otro ejemplo de ambigüedad. Las siguientes versiones sobrecargadas de **vaPrueba( )** son esencialmente ambigua aunque una toma parámetros normales.

```
static void vaPrueba(int … v) { // …

static void vaPrueba(int n, int … v) { // …
```

Aunque las listas de parámetros en cada versión de **vaPrueba( )** difieren, no hay forma de que el compilador resuelva la siguiente llamada:

```
vaPrueba(1)
```

¿Esto se traduce a una llamada a **vaPrueba(int …)**, con un argumento varargs, o bien a una llamada a **vaPrueba(int, int …)** sin argumentos varargs? No hay forma de que el compilador de una respuesta a esta pregunta. Así que la situación es ambigua.

Debido a que los errores de ambigüedad, como los que se mostraron antes, algunas veces será necesario privarse de sobrecargar y simplemente usar dos nombres diferentes para los métodos. En algunos casos, los errores de ambigüedad dejan al descubierto una falla conceptual en el código.

CAPÍTULO 8

Herencia

La herencia es una de las piedras angulares de la programación orientada a objetos, ya que permite la creación de clasificaciones jerárquicas. Mediante la herencia se puede crear una clase general que define rasgos generales para un conjunto de términos relacionados. Esta clase puede ser heredada por otras clases más específicas, cada una de las cuales añadirá aquellos elementos que la distinguen. En la terminología de Java, una clase que es heredada se denomina *superclase*. La clase que hereda se denomina *subclase*. Por lo tanto, una subclase es una versión especializada de una superclase, que hereda todas las variables de instancia y métodos definidos por la superclase y añade sus propios elementos.

Fundamentos de la herencia

Para heredar una clase, simplemente se incorpora la definición de una clase dentro de la otra usando la palabra clave **extends**. Comencemos con un ejemplo corto para explicar esta idea. El siguiente programa crea una superclase denominada **A** y una subclase denominada **B**. La subclase **B** se crea mediante la palabra clave **extends**.

```
// Un ejemplo simple de herencia.

// Creación de la superclase A.
class A {
  int i, j;

  void mostrarij () {
    System.out.println ("i y j: " + i + " " + j);
  }
}

// Creación de la subclase B por extensión de la clase A
class B extends A {
  int k;

  void mostrark () {
    System.out.println ("k: " + k);
  }
  void suma () {
    System.out.println ("i + j + k : " + (i+j+k));
```

```
  }
}

class HerenciaSimple {
  public static void main (String args[]) {
    A superOb = new A() ;
    B subOb = new B();

    // Se puede utilizar la superclase independientemente de sus subclases
    superOb.i = 10;
    superOb.j = 20;
    System.out.println ("Contenido de superOb: ");
    superOb.mostrarij ();
    System.out.println ();

    /* La subclase accede a todos los miembros públicos de
    su superclase. */
    subOb.i = 7;
    subOb.j = 8;
    subOb.k = 9;
    System.out.println ("Contenido de subOb: ");
    subOb.mostrarij ();
    subOb.mostrark ();
    System.out.println ();

    System.out.println ("Suma de i, j y k en subOb:");
    subOb.suma ();
  }
}
```

La salida de este programa es la siguiente:

```
Contenido de superOb:
i y j: 10 20

Contenido de subOb:
i y j: 7 8
k: 9

Suma de i, j y k en subOb:
i + j + k: 24
```

Como se puede comprobar, la subclase **B** incluye todos los miembros de su superclase, **A**. Por esta razón **subOb** puede acceder a **i** y **j** y llamar a **mostrarij( ).** También, dentro de **suma( )** se puede hacer referencia a **i** y **j** directamente como si fueran parte de **B**.

Aunque **A** es una superclase para **B**, es también completamente independiente de **B**. El hecho de ser una superclase para una subclase no implica que no pueda ser utilizada por sí sola. Más aún, una subclase puede ser una superclase para otras subclases.

La forma general de la declaración de una **clase** que hereda una superclase es la siguiente:

class *nombre_subclase* extends *nombre_superclase* {
// cuerpo de la clase
}

Solamente se puede especificar una superclase para cada subclase creada. Java no permite que una subclase herede de múltiples superclases. Se puede crear, como se ha dicho, una

jerarquía de herencia en la que cada subclase se convierta en una superclase para otra subclase. Pero ninguna clase puede ser una superclase de sí misma.

Acceso a miembros y herencia

Aunque una subclase incluye a todos los miembros de su superclase, no puede acceder a aquellos miembros de la superclase que se hayan declarado como **privados**. Por ejemplo, consideremos la siguiente jerarquía de clases:

```
/* En una jerarquía de clases, los miembros privados permanecen
   como privados para su clase.

    Este programa contiene un error
    y no compilará.
*/

// Crear una superclase.
class A {
  int i; // público por defecto
  private int j;  // privado para A

  void setij (int x, int y) {
    i = x;
    j = y;
  }
}

// Aquí no se puede acceder a la variable j de A.
class B extends A {
  int total;
  void suma () {
    total = i + j;  // ERROR, aquí no se puede acceder a j
  }
}

class Access {
  public static void main (String args[]) {
    B subOb = new B();

    subOb. setij (10, 12);

    subOb.suma ();
    System.out.println ("Total es igual a " + subOb. total);
  }
}
```

Este programa no compilará porque la referencia a **j** dentro del método **suma( )** de **B** da lugar a una violación de acceso. Como **j** se declara **private**, solamente tienen acceso a esta variable los miembros de su propia clase. Las subclases no tienen acceso a ella.

NOTA *Un miembro de una clase que ha sido declarado privado, permanece como privado para su clase y no es accesible desde cualquier código que esté fuera de su clase, incluyendo las subclases.*

Un ejemplo más práctico

Veamos a continuación un ejemplo más práctico que sirve para ilustrar el poder de la herencia. Aquí, la versión final de la clase **Caja**, desarrollada en los capítulos precedentes, se extenderá para incluir un cuarto componente denominado **peso**. Así la nueva clase contiene el largo, ancho, alto y peso de la caja.

```
// Este programa utiliza la herencia para extender el programa Caja
class Caja {
  double ancho;
  double alto;
  double largo;

  // constructor para duplicados de un objeto
  Caja (Caja ob) {  // paso del objeto al constructor
    ancho = ob.ancho;
    alto = ob.alto;
    largo = ob.largo;
  }

  // constructor que se utiliza cuando se especifican todas las dimensiones
  Caja (double w, double h, double d) {
    ancho = w;
    alto = h;
    largo = d;
  }

  // constructor que se utiliza cuando no se especifican dimensiones
  Caja () {
    ancho = -1; // usa -1 paran indicar
    alto = -1; // que una caja no está
    largo = -1; // inicializada
  }

  // constructor que se utiliza para crear un cubo
  Caja (double len) {
    ancho = alto = largo = len;
  }

  // cálculo y devolución del volumen
  double volumen () {
    return ancho * alto * largo;
  }
}

// Aquí, se extiende Caja para incluir el peso.
class PesoCaja extends Caja {
    double peso;  // peso de la caja

// constructor para PesoCaja
PesoCaja (double w, double h, double d, double m) {
    ancho = w;
    alto = h;
    largo = d;
    peso = m;
  }
}
```

```
class DemoPesoCaja {
  public static void main (String args[]) {
    PesoCaja miCaja1 = new PesoCaja (10, 20, 15, 34.3);
    PesoCaja miCaja2 = new PesoCaja (2, 3, 4, 0.076);
    double vol;

    vol = miCaja1.volumen();
    System.out.println ("El volumen de miCaja1 es " + vol);
    System.out.println ("El peso de miCaja1 es " + miCajal.peso);
    System.out.println ();

    vol = miCaja2.volumen () ;
    System.out.println ("El volumen de miCaja2 es " + vol);
    System.out.println ("El peso de miCaja2 es " + miCaja2.peso);
  }
}
```

La salida de este programa es la que se muestra a continuación:

```
El volumen de miCajal es 3000.0
El peso de miCajal es 34.3

El volumen de miCaja2 es 24.0
El peso de miCaja2 es 0.076
```

La clase **PesoCaja** hereda todas las características de **Caja** y añade la componente **peso**. La clase **PesoCaja** no necesita volver a crear todas las características que se encuentran en **Caja**; en lugar de ello le basta con extender a la clase **Caja**.

La mayor ventaja de la herencia es que, una vez que se ha creado una superclase, que define los atributos comunes a un conjunto de objetos, se puede utilizar para crear cualquier número de clases más específicas. Cada subclase puede adoptar de forma más precisa su propia clasificación. Por ejemplo, la siguiente clase hereda **Caja** y añade el atributo del color:

```
// En este ejemplo se extiende Caja para incluir el color.
class ColorCaja extends Caja {
  int color;  // color de la caja

  ColorCaja (double w, double h, double d, int c) {
    Ancho = w;
    alto = h;
    largo = d;
    color = c;
  }
}
```

Recuerde que una vez que se ha creado una superclase que define los aspectos generales de un objeto, esa superclase puede ser heredada para formar clases especializadas. Cada subclase simplemente añade sus propios y únicos atributos. Esto es en esencia la herencia.

Una variable de una superclase puede referirse a un objeto de tipo subclase

Se puede asignar a una variable de referencia de una superclase una referencia a cualquier subclase derivada de esa superclase. Este aspecto de la herencia resulta muy útil en diferentes situaciones. Consideremos el siguiente ejemplo:

```
class RefDemo {
  public static void main (String args[]) {
    PesoCaja pesoCaja = new PesoCaja (3, 5, 7, 8.37);
    Caja cajaSencilla = new Caja ();
    double vol;

    vol = pesoCaja.volumen ();
    System.out.println ("El volumen de pesoCaja es " + vol);
    System.out.println ("El peso de pesoCaja es " +
                        pesoCaja.peso);
    System.out.println ();

    // se asigna una referencia de PesoCaja a una referencia de Caja
    cajaSencilla = pesoCaja;

    vol = cajaSencilla.volumen(); // Es correcto, ya que volumen()
    está definido en Caja
    System.out.println ("El volumen de cajaSencilla es " + vol);

    /* La siguiente sentencia no es válida ya que
       cajaSencilla no define como miembro a peso. */
// System.out.println ("El peso de cajaSencilla es " + cajaSencilla.peso);
  }
}
```

Aquí, **pesoCaja** es una referencia a un objeto de la clase **PesoCaja**, y **cajaSencilla** es una referencia a un objeto de la clase **Caja**. Como **PesoCaja** es una subclase de Caja, se permite asignar a **cajaSencilla** una referencia a objetos **pesoCaja**.

Es importante comprender que es el tipo de la variable de referencia y no el tipo de objeto al que se refiere, lo que determina a qué miembros se puede acceder; es decir, cuando una referencia a un objeto subclase se asigna a una variable de referencia de la superclase, se tendrá acceso sólo a aquellas partes del objeto definidas por la superclase. Éste es el motivo por el cual **cajaSencilla** no puede acceder a **peso**, aunque se refiera a un objeto **PesoCaja**. Esto es lógico, ya que la superclase no tiene conocimiento de lo que la subclase añade a su definición, y a ello se refiere el comentario de la última línea del fragmento de código anterior. No es posible para una referencia a un objeto de la clase **Caja** acceder al campo **peso**, ya que este campo no está definido en la clase **Caja**.

Aunque los comentarios anteriores puedan resultar un tanto extraños, tienen importantes aplicaciones, dos de las cuales se comentarán más adelante, en este capítulo.

super

En los ejemplos anteriores las clases obtenidas a partir de **Caja** no fueron implementadas tan eficiente o robustamente como podrían haberlo sido. Por ejemplo, el constructor **PesoCaja** inicializa explícitamente los campos de **Caja( )**, **ancho**, **largo** y **alto**. Este código es ineficiente, y no sólo se encuentra duplicado en su superclase, sino que además implica que el acceso a estos miembros de la subclase debe ser garantizado. Sin embargo, puede haber ocasiones en las que se desee crear una superclase que mantenga para sí misma los detalles de su implementación,

es decir, que mantenga privados sus datos miembros. En este caso, la subclase no podrá acceder directamente o inicializar estas variables. Ya que la encapsulación es un atributo primario de la programación orientada a objetos, no sorprende que Java proporcione la solución a este problema. Siempre que una subclase necesite referirse a su superclase inmediata, se utilizará la palabra clave **super**.

La palabra clave **super** tiene dos formas generales. La primera llama al constructor de la superclase. La segunda se usa para acceder a un miembro de la superclase que ha sido escondido por un miembro de una subclase. A continuación se examina cada uno de estos usos.

Usando super para llamar a constructores de superclase

Una subclase puede llamar a un método constructor definido por su superclase utilizando la siguiente forma de **super**:

super (*lista de parámetros*);

La *lista de parámetros* especifica cualquier parámetro que el constructor necesite en la superclase. **super( )** debe ser siempre la primera sentencia que se ejecute dentro de un constructor de la subclase.

Para ver cómo se usa **super( )**, consideremos esta versión mejorada de la clase **PesoCaja**

```
// PesoCaja utiliza ahora super para inicializar los atributos de Caja.
class PesoCaja extends Caja {
  double peso; // peso de la caja

  // Inicialización de ancho, largo y alto usando super()
  PesoCaja (double w, double h, double d, double m) {
    super (w, h, d); // llamada al constructor de la superclase
    peso = m;
  }
}
```

En este ejemplo, **PesoCaja( )** llama a **super( )** con los parámetros **w**, **h** y **d**. Esto hace que se ejecute el constructor de **Caja( )** que inicializa las variables **ancho**, **largo** y **alto**. **PesoCaja** ya no tiene que inicializar estos valores; sólo necesita inicializar un único valor, el **peso**. Esto permite a **Caja** definir estos valores privados si así lo desea.

En el ejemplo anterior, se ha llamado a **super( )** con tres argumentos. Como los constructores pueden ser sobrecargados, se puede llamar a **super( )** usando cualquier forma definida por la superclase. El constructor que se ejecute será el que presente coincidencia de parámetros. El ejemplo que se presenta a continuación es una implementación completa de **PesoCaja** que proporciona los constructores correspondientes a las diferentes formas en que se puede construir una caja. En cada caso, se llama a **super( )** utilizando los argumentos apropiados. Observe que se han hecho privadas las variables **ancho**, **largo** y **alto** dentro de **Caja**.

```
// Implementación completa de PesoCaja.
class Caja {
  private double ancho;
  private double alto;
  private double largo;
```

```
  // constructor para duplicados de un objeto
  Caja (Caja ob) {  // se pasa el objeto al constructor
    ancho = ob.ancho;
    alto = ob.alto;
    largo = ob.largo;
  }

  // constructor usado cuando se especifican todas las dimensiones
  Caja (double w, double h, double d) {
    ancho = w;
    alto = h;
    largo = d;
  }

  // constructor usado cuando no se especifican dimensiones
  Caja () {
    ancho = -1; // usa -1 para indicar que
    alto = -1;  // la caja no está
    largo = -1; // inicializada
  }

  // constructor usado cuando se crea un cubo
  Caja (double len) {
    ancho = alto = largo = len;
  }

  // se calcula y devuelve el volumen
  double volumen () {
    return ancho * alto * largo;
  }
}

// PesoCaja ahora implementa completamente todos los constructores.
class PesoCaja extends Caja {
  double peso;  // peso de la caja

// constructor para duplicados de un objeto
    PesoCaja (PesoCaja ob) {  // se pasa el objeto al constructor
    super (ob);
    peso = ob.peso;
  }

  // constructor que se utiliza cuando se especifican todos los parámetros
  PesoCaja(double w, double h, double d, double m) {
    super (w, h, d);  // llamada al constructor de la superclase
    peso = m;
  }

  // constructor por defecto
    PesoCaja () {
    super ();
    peso = -1;
  }

  // constructor que se utiliza cuando se crea un cubo
  PesoCaja (double len, double m) {
```

```
    super (len) ;
    peso = m;
  }
}

class DemoSuper {
  public static void main (String args[]) {
    PesoCaja miCaja1 = new PesoCaja (10, 20, 15, 34.3);
    PesoCaja miCaja2 = new PesoCaja (2, 3, 4, 0.076);
    PesoCaja miCaja3 = new PesoCaja (); // por omisión
    PesoCaja miCubo = new PesoCaja (3, 2);
    PesoCaja miDup1icado = new PesoCaja (miCaja1);
    double vol;

    vol = miCajal.volumen();
    System.out.println ("El volumen de miCajal es " + vol);
    System.out.println ("El peso de miCajal es " + miCajal.peso);
    System.out.println ();

    vol = miCaja2.volumen();
    System.out.println ("El volumen de miCaja2 es " + vol);
    System.out.println ("El peso de miCaja2 es " + miCaja2.peso);
    System.out.println ();

    vol = miCaja3.volumen();
    System.out.println ("El volumen de miCaja3 es " + vol);
    System.out.println ("El peso de miCaja3 es " + miCaja3.peso);
    System.out.println ();

    vol = miDuplicado.volumen();
    System.out.println ("El volumen de miDuplicado es " + vol);
    System.out.println ("El peso de miDuplicado es " + miDuplicado.peso);
    System.out.println ();

    vol = miCubo.volumen();
    System.out.println ("El volumen de miCubo es " + vol);
    System.out.println ("El peso de miCubo es " + miCubo.peso);
    System.out.println ();
  }
}
```

Este programa genera la siguiente salida:

```
El volumen de miCaja1 es 3000.0
El peso de miCaja1 es 34.3

El volumen de miCaja2 es 24.0
El peso de miCaja2 es 0.076

El volumen de miCaja3 es -1.0
El peso de miCaja3 es -1.0

El volumen de miDuplicado es 3000.0
El peso de miDuplicado es 34.3

El volumen de miCubo es 27.0
El peso de miCubo es 2.0
```

Prestemos especial atención al siguiente constructor **PesoCaja( )**:

```
// se construye un duplicado de un objeto
PesoCaja (PesoCaja ob) {   // se pasa el objeto al constructor
  super (ob);
  peso = ob.peso;
}
```

En este caso se llama a **super( )** con un objeto del tipo **PesoCaja** —no del tipo **Caja**— y sirve para llamar al constructor **Caja(Caja ob).** Como se mencionó anteriormente se puede utilizar una variable de la superclase para referenciar cualquier objeto derivado de esa clase. De esa manera, podemos pasar un objeto **PesoCaja** al constructor de **Caja**. Evidentemente, **Caja** sólo tiene información de sus propios miembros.

Repasemos los conceptos más importantes relativos a **super( ).** Cuando una subclase llama a **super( )**, está llamando al constructor de su superclase inmediata. Así, **super( )** siempre se refiere a la superclase inmediatamente superior a la clase llamante. Esto se cumple incluso en una jerarquía con múltiples niveles. Además, **super( )** debe ser la primera sentencia que se ejecute dentro del constructor de una subclase.

Un segundo uso de super

La segunda forma de **super** actúa de una forma parecida a **this**, excepto que siempre se refiere a la superclase de la subclase en la que se usa. Este uso tiene la siguiente forma general:

super.*miembro*

En este caso, *miembro* puede ser un método o una variable de instancia.

Esta segunda forma de **super** tiene una mayor aplicación en situaciones en las que los nombres de miembros de una subclase ocultan miembros del mismo nombre en la superclase. Consideremos la siguiente jerarquía de clases:

```
// Uso de super para evitar el ocultamiento de nombres.
class A {
  int i;
}

// Se crea una subclase extendiendo la clase A.
class B extends A {
  int i; //esta variable i oculta la variable i de A

  B (int a, int b) {
    super.i = a; // i de A
    i = b;  // i de B
  }

  void show() {
    System.out.println ("i en la superclase: " + super.i);
    System.out.println ("i en la subclase: " + i);
  }
}

  class UsaSuper {
    public static void main (String args[]) {
    B subOb = new B (1, 2);
```

```
    subOb.show ();
  }
}
```

Este programa presenta la siguiente salida:

```
i en la superclase: 1
i en la subclase: 2
```

Aunque la variable de instancia **i** de **B** oculta la **i** de **A**, **super** permite acceder a la **i** definida en la superclase. Como se puede observar, **super** también se puede utilizar para llamar a métodos ocultos por una subclase.

Creación de una jerarquía multinivel

Hasta este momento, hemos estado utilizando jerarquías de clases sencillas que consisten sólo en una superclase o una subclase. Sin embargo, es posible construir jerarquías que contengan tantos niveles de herencia como se quiera. Como se ha mencionado, es perfectamente aceptable que una subclase sea la superclase de otra clase. Por ejemplo, las clases **A**, **B**, **C**, donde **C** puede ser una subclase de **B**, que a su vez es una subclase de **A**. Cuando se produce esta situación, cada subclase hereda todos atributos encontrados en todas sus superclases. En este caso, **C** hereda todas las características de **B** y **A**. Consideremos el siguiente programa para ver cómo se puede utilizar una jerarquía multinivel. En el ejemplo, la subclase **PesoCaja** se utiliza como una superclase para crear una subclase denominada **Envio**. La clase **Envio** hereda todos los atributos de **PesoCaja** y **Caja**, y añade un campo denominado costo, que contiene el costo del envío por paquete.

```
// Se extiende PesoCaja para incluir el costo del envío.

// Comienzo con Caja.
class Caja {
    private double ancho;
    private double alto;
    private double largo;

  // construye un duplicado de un objeto
  Caja (Caja ob) {  // se pasa el objeto al constructor
    ancho = ob.ancho;
    alto = ob.alto;
    largo = ob.largo;
  }

  // constructor usado cuando se especifican todas las dimensiones
  Caja (double w, double h, double d) {
    ancho = w;
    alto = h;
    largo = d;
  }

  // constructor usado cuando no se especifican dimensiones
  Caja () {
    ancho = -1; // usa -1 para indicar
    alto = -1;  // que la caja no está
```

```
    largo = -1; // inicializada
  }

  // constructor usado cuando se crea un cubo
  Caja (double lon) {
    ancho = alto = largo = lon;
  }

  // se calcula y devuelve el volumen
    double volumen () {
    return ancho * alto * largo;
  }
}

  // Se añade el peso.
  class pesoCaja extends Caja {
    double peso;  // peso de la caja

  // se construye una copia de un objeto
  PesoCaja (PesoCaja ob) {   // se pasa el objeto al constructor
    super (ob) ;
    peso = ob.peso;
  }

  // constructor que se utiliza cuando se especifican todos los parámetros
  PesoCaja (double w, double h, double d, double m) {
    super (w, h, d);  // llamada al constructor de la superclase
    peso = m;
  }

  // constructor por omisión
  PesoCaja () {
    super ();
    peso = -1;
  }

  // constructor usado cuando se crea un cubo
  PesoCaja (double lon, double m) {
    super (lon);
    peso = m;
  }
}

  // Se añaden los costos del envío
  class Envio extends PesoCaja {
    double costo;

    // construye un duplicado de un objeto
    Envio (Envio ob) {   // se pasa el objeto al constructor
      super (ob) ;
      costo = ob.costo;
    }

    // constructor cuando se especifican todos los parámetros
    Envio (double w, double h, double d,
            double m, double c) {
```

```
    super (w, h, d, m);  // llamada al constructor de la superclase
    costo = c;
  }

    // constructor por omisión
    Envio () {
      super ();
      costo = -1;
  }

  // constructor que se utiliza cuando se crea un cubo
  Envio (double lon, double m, double c) {
    super (lon, m);
    costo = c;
  }
}

class DemoEnvio {
  public static void main (String args[]) {
    Envio envio1 =
             new Envio (10, 20, 15, 10, 3.41);
    Envio envio2 =
             new Envio (2, 3, 4, 0.76, 1.28);

    double vol;

    vol = envio1.volumen ();
    System.out.println ("EI volumen del envio1 es " + vol);
    System.out.println ("EI peso del envio1 es " +
                        envio1.peso);
    System.out.println ("Costo del envío1 es: $ " + envio1.costo);
    System.out.println ();

    vol = envio2.volumen();
    System.out.println ("EI volumen del envio2 es " + vol);
    System.out.println ("EI peso del envio2 es "
                        + envio2. peso) ;
    System.out.println ("Costo del envio2 es: $ " + envio2.costo);
  }
}
```

La salida de este programa es:

```
El volumen del envio1 es 3000.0
El peso del envio1es 10.0
Costo del envio1 es: $ 3.41

El volumen del envio2 es 24.0
El peso del envio2 es 0.76
Costo del envio2: $ 1.28
```

Gracias a la herencia, la clase **Envio** puede utilizar las clases **Caja** y **PesoCaja** definidas previamente, añadiendo solamente la información adicional que necesita para su aplicación específica. Esta es una de las ventajas de la herencia, permite la reutilización del código.

Este ejemplo pone de manifiesto otro punto importante: **super( )** siempre se refiere al constructor de la superclase más próxima. El método **super( )** de la clase **Envio** llama al constructor de **PesoCaja**. El método **super( )** en **PesoCaja** llama al constructor de **Caja**. En una jerarquía de clases, si el constructor de una superclase requiere parámetros, entonces todas las subclases deben pasar esos parámetros "hacia arriba". Esto debe ser así, sin importar que la subclase necesite o no esos parámetros.

NOTA *En el programa anterior, la jerarquía de clases, que contiene Caja, PesoCaja y Envio, se incluye en un único archivo, pero no tiene por qué ser así. En Java, cada una de las tres clases podría haberse colocado en su propio archivo y compilado por separado. De hecho, la utilización de archivos distintos es la norma y no la excepción en la creación de una jerarquía de clases.*

Cuándo son ejecutados los constructores

Cuando se crea una jerarquía de clases, ¿en qué orden se ejecutan los constructores de las clases que forman la jerarquía? Por ejemplo, dada una subclase denominada **B** y una superclase denominada **A**, ¿se ejecuta el constructor de **A** antes o después que el constructor de **B**? La respuesta es que, en una jerarquía de clases, los constructores se ejecutan en el orden en que se derivan, desde la superclase a la subclase. Además, ya que **super( )** debe ser la primera sentencia que se ejecute en el constructor de una subclase, este orden es el mismo tanto si se usa **super( )**, como si no se usa. Si no se utiliza **super( )**, se ejecuta el constructor por defecto o constructor sin parámetros de cada superclase. El siguiente programa muestra cuándo se ejecutan los constructores:

```
// Demuestra cuándo se ejecutan los constructores.

// Se crea una superclase.
class A {
  A () {
    System.out.println ("Dentro del constructor de A.");
  }
}

// Se crea una subclase extendiendo la clase A.
class B extends A {
  B () {
    System.out.println ("Dentro del constructor de B.");
  }
}

// Se crea otra subclase C extendiendo B.
class C extends B {
  C () {
    System.out.println ("Dentro del constructor de C.");
  }
}

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

```
    C c = new C();
  }
}
```

La salida de este programa es la siguiente:

```
Dentro del constructor de A
Dentro del constructor de B
Dentro del constructor de C
```

Como se puede observar, los constructores se ejecutan en el orden en que se derivan.

Este orden de ejecución de las funciones del constructor es lógico, ya que una superclase no tiene conocimiento de sus subclases, y cualquier inicialización que necesite es independiente, y posiblemente un prerrequisito para cualquier inicialización realizada por la subclase, por lo que se debe ejecutar en primer lugar.

Sobrescritura de métodos

En una jerarquía de clases, cuando un método de una subclase tiene el mismo nombre y tipo que un método de su superclase, entonces se dice que el método de la subclase sobrescribe al método de la superclase. Cuando se llama a un método sobrescrito desde una subclase, esta llamada siempre se refiere a la versión de ese método definida por la subclase. La versión del método definida por la superclase queda oculta. Consideremos el siguiente ejemplo:

```
// Sobreescritura de métodos.
class A {
  int i, j;
  A (int a, int b) {
    i = a;
    j = b;
  }

  // se imprimen i y j
  void show() {
    System.out.println ("i y j: " + i + " " + j);
  }
}

class B extends A {
  int k;

  B (int a, int b, int c) {
    super (a, b);
    k = c;
  }

  // se imprime k - sobrescribiendo el método show() en A
  void show() {
    System.out.println ("k: " + k);
  }
}
```

```
class Sobreescribe {
  public static void main (String args[]) {
    B subOb = new B(1, 2, 3);

    subOb.show();  // llamada a show() en B
  }
}
```

La salida que produce este programa es la siguiente:

```
k: 3
```

Cuando se invoca a **show( )** en un objeto del tipo **B**, se utiliza la versión de **show( )** definida dentro de **B**; es decir, la versión de **show( )** dentro de **B** sobrescribe a la versión declarada en **A**.

Si se desea acceder al método sobrescrito de la superclase, es posible hacerlo utilizando **super**. Por ejemplo, en esta versión de **B**, se llama a la versión de **show( )** de la superclase dentro de la versión de la subclase. Esto permite imprimir todas las variables de instancia.

```
class B extends A {
  int k:

  B (int a, int b, int c) {
    super (a, b);
    k = c;
  }
  void show() {
    super.show(); // llamada al método show() de A.
    System.out.println ("k: " + k):
  }
}
```

Sustituyendo la versión de **A** del programa anterior por esta versión, se obtiene la siguiente salida:

```
i y j: 1 2
k: 3
```

Aquí, **super.show( )** llama a la versión de **show( )** de la superclase.

La sobreescritura de métodos aparece *únicamente* cuando los nombres y tipos de los dos métodos son idénticos. Si no lo son, entonces los dos métodos están simplemente sobrecargados. Consideremos, por ejemplo, la siguiente versión modificada del ejemplo anterior.

```
// Los métodos con diferentes tipos se sobrecargan,
no se sobrescriben.
class A {
  int i, j;

  A (int a, int b) {
    i = a;
    j = b;
  }

  // se imprime i y j
  void show () {
```

```
    System.out.println ("i y j: " + i + " " + j);
  }
}

// Creación de una subclase extendiendo la clase A.
class B extends A {
  int k;

  B (int a, int b, int c) {
    super (a, b);
    k = c;
  }

  // sobrecarga de show ()
  void show (String msg) {
    System.out.println (msg + k);
  }
}

class Sobreescribe {
  public static void main (String args[]) {
    B subOb = new B(1, 2, 3);

    subOb.show ("Esto es k: "); // llamada a show() de B
    subOb.show (); // llamada a show () de A
  }
}
```

La salida generada por este programa es la siguiente:

```
Esto es k: 3
i y j: 1 2
```

La versión de **show( )** en **B** tiene un parámetro de tipo String lo cual hace que su firma sea diferente del método **show( )** en **A**, que no tiene parámetros. Por ello, no existe sobreescritura u ocultamiento del nombre. En su lugar, la versión de **show( )** en **B** simplemente sobrecarga la versión de **show( )** en **A**.

Selección dinámica de métodos

Si bien los ejemplos de la sección precedente demuestran el mecanismo de sobrescritura de métodos, no muestran realmente todas sus posibilidades. En efecto, si dichos métodos no fueran más que un convenio del espacio de nombres, entonces serían una curiosidad interesante pero de poco valor práctico. Sin embargo, esto no es así. La sobrescritura de métodos es la base de uno de los conceptos más potentes de Java: la *selección dinámica de métodos*, mediante este mecanismo la llamada a una función sobrescrita se resuelve en el tiempo de ejecución y no en el tiempo de compilación. La selección dinámica de métodos tiene una gran importancia en Java, ya que permite implementar el polimorfismo durante el tiempo de ejecución.

Comencemos a plantear un importante principio: una variable de referencia de una superclase se puede referir a un objeto de una subclase. Java se basa en esto para resolver

llamadas a métodos sobrescritos en el tiempo de ejecución. Cuando se llama a un método sobrescrito a través de una referencia a una superclase, Java determina qué versión de ese método se debe ejecutar en función del tipo de objeto referido cuando se produce la llamada. Por lo tanto, esta determinación se produce en el tiempo de ejecución. Cuando se hace referencia a diferentes tipos de objetos, se llama a diferentes versiones de métodos sobrescritos. En otras palabras, *lo que determina la versión del método sobrescrito* que será ejecutado es el tipo de objeto al que se hace referencia y no el tipo de variable de referencia. Por lo tanto, si una superclase contiene un método sobrescrito por una subclase, entonces cuando se haga referencia a distintos tipos de objetos mediante una variable de referencia de una superclase, se ejecutarán diferentes versiones del método.

El siguiente ejemplo ilustra la selección dinámica de métodos.

```
// Selección dinámica de métodos.
class A {
  void callme () {
    System.out.println ("Llama al método callme, dentro de A");
  }
}

class B extends A {
  // sobrescribe callme ()
  void callme () {
    System.out.println ("Llama al método callme, dentro de B");
  }
}

class C extends A {
  // sobrescribe callme ()
  void callme () {
    System.out.println ("Llama al método callme, dentro de C");
  }
}

class Dispatch {
  public static void main (String args[]) {
    A a = new A(); // objeto del tipo A
    B b = new B(); // objeto del tipo B
    C c = new C(); // objeto del tipo C
    A r; // una referencia del tipo A

    r = a; // r se refiere a un objeto A
    r.callme ();  // llamada a la versión de callme en A

    r = b  // r se refiere a un objeto B
    r.callme ();  // llamada a la versión de callme en B

    r = c// r se refiere a un objeto e
    r.callme ();  // llamada a la versión de callme en C
  }
}
```

La salida de este programa es la siguiente:

```
Llama al método callme, dentro de A
Llama al método callme, dentro de B
Llama al método callme, dentro de C
```

Este programa crea una superclase llamada **A** y dos subclases de la misma, llamadas **B** y **C**. Las subclases **B** y **C** sobreescriben el método **callme( )** declarado en **A**. Los objetos del tipo **A**, **B** y **C** se declaran dentro del método **main( )**. También se declara una referencia del tipo **A** denominada **r**. El programa asigna entonces una referencia de cada tipo de objeto a la variable **r** y utiliza esa referencia para invocar a **callme( ).** Como muestra la salida, la versión de **callme( )** que se ejecuta queda determinada por el tipo de objeto al que se hace referencia en el instante en que se produce la llamada.

Si hubiera sido determinada por el tipo de la variable de referencia **r**, se hubieran producido tres llamadas al método **callme( )** de **A**.

NOTA *Los lectores familiarizados con C++ o C# reconocerán las similitudes entre los métodos sobrescritos de Java y las funciones virtuales de esos lenguajes.*

¿Por qué se sobrescriben los métodos?

Como se ha comentado anteriormente, los métodos sobrescritos permiten que Java soporte el polimorfismo en tiempo de ejecución. El polimorfismo es esencial en la programación orientada a objetos, porque permite que una clase general especifique métodos que serán comunes a todas las clases que se deriven de la misma; de manera que las subclases podrán definir la implementación de algunos o todos esos métodos. Los métodos sobrescritos son otra de las formas en que Java implementa el aspecto del polimorfismo que se puede expresar como"una interfaz, múltiples métodos".

Para aplicar satisfactoriamente el polimorfismo es importante comprender que las superclases y las subclases forman una estructura jerárquica que se mueve desde una menor a una mayor especialización. Cuando se utiliza correctamente, una superclase proporciona todos los elementos que una subclase puede usar directamente. También define aquellos métodos que las clases que se deriven de ella deben implementar por sí mismas. Esto da a la subclase la flexibilidad de definir sus propios métodos y al mismo tiempo asegura una interfaz consistente. De esta forma, combinando la herencia con los métodos sobrescritos, una superclase puede definir la forma general de los métodos que se usarán en todas sus subclases.

El polimorfismo dinámico en tiempo de ejecución es uno de los mecanismos más potentes que aporta la programación orientada a objetos para conseguir la reutilización del código y la robustez. La posibilidad de utilizar bibliotecas de códigos existentes para llamar a métodos en instancias de nuevas clases sin volver a compilar, a la vez que se mantiene una interfaz abstracta, es una herramienta profundamente poderosa.

Aplicación de la sobrescritura de métodos

Veamos un ejemplo más práctico en el que se utiliza la sobrescritura de métodos. El siguiente programa crea una superclase denominada **Figura** que almacena las dimensiones de un objeto bidimensional. También define un método llamado **area( )**, que calcula el área del objeto. El programa deriva dos subclases de **Figura**. La primera es **Rectangulo** y la segunda **Triangulo**.

Cada una de estas subclases sobrescribe **area( )** para devolver el área del rectángulo y del triángulo respectivamente.

```
// Usando polimorfismo en tiempo de ejecución.
class Figura {
  double dim1;
  double dim2;

  Figura (double a, double b) {
    diml = a;
    dim2 = b;
  }

  double area () {
    System.out.println ("El área de la figura no está definida.");
    return ();
  }
}

class Rectangulo extends Figura {
  Rectangulo (double a, double b) {
    super (a, b);
  }

  // sobrescribe el método área para un rectángulo
  double area () {
    System.out.println ("Dentro del método área para un objeto rectángulo.");
    return diml* dim2;
  }
}

class Triangulo extends Figura {
  Triangulo (double a, double b) {
    super (a, b);
  }

  //sobrescribe el método área para un triángulo rectángulo
  double area () {
    System.out.println ("Dentro del método área para un objeto triángulo.");
    return dim1 * dim2 / 2;
  }
}

class CalculoAreas {
  public static void main (String args[]) {
    Figura f = new Figura (10, 10);
    Rectangulo r = new Rectangulo (9, 5);
    Triangulo t = new Triangulo (10, 8);
    Figura figref;

    figref = r;
    System.out.println ("El área es " + figref.area());

    figref = t;
    System.out.println ("El área es " + figref.area());
```

```
    figref = f;
    System.out.println ("El área es " + figref.area());
  }
}
```

La salida de este programa es la siguiente:

```
Dentro del método área para un objeto rectángulo.
El área es 45
Dentro del método área para un objeto triángulo.
El área es 40
El área de la figura no está definida.
El área es 0
```

Por medio del mecanismo dual de la herencia y el polimorfismo en tiempo de ejecución es posible definir una interfaz consistente que puede ser utilizada por objetos de tipos distintos, aunque relacionados. En el ejemplo anterior, si un objeto se deriva de **Figura**, su área se puede obtener llamando al método **area( )**. La interfaz para esta operación es la misma, independientemente de cuál sea el tipo de figura que se esté usando.

Clases abstractas

Hay ocasiones en las que se quiere definir una superclase en la que se declare la estructura de una determinada abstracción sin implementar completamente cada método, es decir, en la que sólo se defina una forma generalizada que será compartida por todas las subclases, dejando que cada subclase complete los detalles necesarios. Una clase de este tipo determina la naturaleza de los métodos que las subclases deben implementar. Un caso en el que se puede producir esta situación es aquel en que una superclase no es capaz de crear una implementación de un método que tenga un significado completo. Esto era precisamente lo que ocurría en la clase **Figura** del ejemplo anterior. La definición de **area( )** era simplemente un patrón, y en la misma no se calculaba ni se imprimía el área de ningún objeto.

Como verá cuando cree sus propias bibliotecas de clases, es habitual que un método no tenga una definición completa dentro de su superclase. Esta situación se puede gestionar de dos formas. Una, tal y como muestra el ejemplo anterior, es, simplemente, presentar un mensaje de aviso. Esto, que puede ser apropiado en determinadas situaciones, como la fase de depuración, no es, normalmente, lo más adecuado. Se pueden definir métodos que deban ser sobrescritos por la subclase para tener un significado completo. Consideremos, por ejemplo, la clase **Triangulo**. Esta clase no tiene significado si no se ha definido el método **area( )**. En este caso, hay que asegurar que una subclase sobrescribe efectivamente todos los métodos necesarios. La solución que da Java a este problema son los *métodos abstractos.*

Se puede precisar que las subclases sobrescriban ciertos métodos especificando el modificador del tipo **abstract**. Se suele hacer referencia a estos métodos como *métodos de responsabilidad* de la subclase, ya que no están implementados específicamente en la superclase. Así, la subclase debe sobrescribirlos, ya que no se puede utilizar la versión de la superclase. Para declarar un método abstracto se utiliza la expresión general:

abstract *tipo-nombre (lista_de_parametros)*;

Como se puede ver, este método no tiene cuerpo.

Cualquier clase que contenga uno o más métodos abstractos debe ser declarada abstracta. Para declarar una clase abstracta, se utiliza la palabra clave **abstract** delante de la palabra clave **class** en el comienzo de la declaración de la clase. No puede haber objetos de clase abstracta, es decir, no se pueden crear instancias de dichas clases directamente con el operador **new**. Tales objetos no tendrían ninguna utilidad, ya que una clase abstracta no está completamente definida. Tampoco se pueden declarar constructores abstractos o métodos estáticos abstractos. Cualquier subclase de una clase abstracta debe implementar todos los métodos abstractos de la superclase, o bien ser declarada ella misma como **abstracta**.

A continuación se presenta un ejemplo sencillo de una clase con un método abstracto, seguido por una clase que implementa el método:

```
// Un ejemplo sencillo de una clase abstracta.
abstract class A {
  abstract void callme ();

  // en las clases abstractas se permiten métodos concretos
  void callmetoo() {
    System.out.println ("Esto es un método concreto.");
  }
}

class B extends A {
  void callme () {
    System.out.println ("Implementación del método callme en B.");
  }
}

class AbstractDemo {
  public static void main(String args[]) {
    B b = new B();

    b.callme();
    b.callmetoo();
  }
}
```

Observe que, en el programa, no se declaran objetos de la clase **A**, ya que no es posible crear una instancia de una clase abstracta. Otro punto de interés es que la clase **A** implementa un método concreto **callmetoo( )**. Esto es perfectamente aceptable, ya que las clases abstractas pueden incluir tanta implementación como sea necesario.

Aunque no se pueden crear instancias de las clases abstractas, sí es posible utilizarlas para crear referencias a objetos, ya que el polimorfismo de Java en tiempo de ejecución se implementa mediante la referencia a las superclases. Por ello, se puede crear una referencia a una clase abstracta de forma que se pueda utilizar como referencia a un objeto de una subclase. En el siguiente ejemplo se muestra esta característica.

Mediante la clase abstracta se puede mejorar la clase **Figura** presentada anteriormente. Dado que el concepto de área no tiene significado para una figura dimensional que no está definida, la siguiente versión del programa declara el método **area( )** como abstracto dentro de **Figura**. Obviamente, esto significa que todas las clases derivadas de **Figura** deben sobrescribir **area( ).**

```
// Uso de métodos y clases abstractas.
abstract class Figura {
  double dim1;
  double dim2;

  Figura (double a, double b) {
    dim1 = a;
    dim2 = b;
  }

  // área es ahora un método abstracto
  abstract double area();
}

class Rectangulo extends Figura {
  Rectangulo (double a, double b) {
    super (a, b);
  }

  // se sobrescribe área para un rectángulo
  double area() {
    System.out.println ("Dentro del método área par un objeto rectángulo.");
    return dim1 * dim2;
  }
}

class Triangulo extends Figura {
  Triangulo (double a, double b) {
    super (a, b);
  }

  //se sobrescribe área para un triángulo
  double area() {
    System.out.println ("Dentro del método área par un objeto triángulo.");
    return dim1 * dim2 / 2;
  }
}

class AbstractAreas {
  public static void main{String args[]) {
  // Figura f = new Figura (10, 10); // ahora esto ya no es correcto
    Rectangulo r = new Rectangulo (9, 5);
    Triangulo t = new Triangulo (10, 8);
    Figura figref;  //esto es correcto, no se crea ningún objeto

    figref = r;
    System.out.println ("El área es " + figref.area());

    figref = t;
    System.out.println ("El área es " + figref.area());
  }
}
```

Tal y como indica el comentario dentro del método **main( )** ya no es posible declarar objetos del tipo **Figura**, porque ahora esta clase es abstracta, y todas sus subclases deben sobrescribir el método **area( ).** Esto se puede probar intentando crear una subclase que no sobrescriba **area( ).** Se obtendrá como respuesta un error del compilador.

Aunque no es posible crear un objeto del tipo **Figura**, sí es posible crear una variable de referencia del tipo **Figura**. La variable **figref** se declara como una referencia a **Figura**, lo que significa que se puede utilizar como referencia a un objeto de cualquier clase derivada de **Figura**. Como se explicó antes, los métodos sobrescritos se resuelven mediante las variables de referencia de la superclase en tiempo de ejecución.

Uso del modificador final con herencia

La palabra clave **final** tiene tres usos. En primer lugar, tal y como se describió en el capítulo anterior, se utiliza para crear el equivalente a una constante con nombre. Los otros dos usos de **final** se aplican a la herencia y se examinan a continuación.

Uso del modificador final para impedir la sobrescritura

La sobreescritura de métodos es una de las características más importantes de Java, pero pueden presentarse ocasiones en las que haya que evitarla. Para imposibilitar que un método sea sobrescrito hay que especificar el modificador **final** en el comienzo de su declaración. Los métodos que se declaran como **final** no pueden ser sobrescritos. El siguiente fragmento de código es un ejemplo de este uso de **final**:

```
class A {
  final void meth () {
    System.out.println ("Este es un método final.");
  }
}

class B extends A {
  void meth() { // ERROR! No está permitida la sobrescritura.
    System.out.println ("No es correcto!");
  }
}
```

Dado que se ha declarado **meth( )** como **final**, no puede ser sobrescrito en **B**. Si se intenta la sobreescritura, el resultado es un error de compilación.

Los métodos declarados como **final** en ocasiones pueden proporcionar una mejora del rendimiento, porque el compilador puede *realizar* llamadas en línea a dichos métodos ya que "sabe" que no pueden ser sobreescritos por una subclase. A menudo, cuando se llama a una pequeña función final, el compilador Java puede copiar el código binario de la subrutina directamente en línea con el código compilado del método llamante, eliminando, por tanto, el trabajo adicional asociado a la llamada de un método. Ésta es una opción de la que disponen solamente los métodos declarados como **final**. Normalmente, Java resuelve las llamadas a métodos dinámicamente en tiempo de ejecución. Esto se suele denominar *asociación tardía (late binding es su nombre en inglés).* Sin embargo, ya que los métodos **final** no pueden ser sobrescritos, una llamada a un método de este tipo se resuelve en el tiempo de compilación. A esto se le denomina *asociación temprana (early binding).*

Uso del modificador final para evitar la herencia

En ocasiones puede ser necesario evitar que una clase sea heredada. Para ello basta con que el nombre de la clase vaya precedido de la palabra clave **final**. Al declarar una clase como **final** se declara también, implícitamente, a todos sus métodos como **final**. Evidentemente no es

válido declarar una clase simultáneamente como **abstract** y **final**, ya que una clase abstracta está incompleta por definición y se basa en sus subclases para proporcionar implementaciones completas.

A continuación se presenta un ejemplo de una clase **final**:

```
final class A {
  // ...
}

// La siguiente clase no es válida.
class B extends A {  // ¡ERROR!, no puede haber subclases de A
  // ...
}
```

Tal y como se ha dicho, es ilegal que la clase **B** herede la clase **A**, ya que la clase **A** se ha declarado como **final**.

La clase object

La clase **Object** es una clase especial, definida por Java. Todas las demás clases de Java son subclases de **Object**, es decir, **Object** es una superclase para todas las demás clases. Esto significa que una variable de referencia del tipo **Object** puede referirse a un objeto de cualquier otra clase. Como los arreglos se implementan como clases, una variable del tipo **Object** puede también referirse a cualquier arreglo.

Object define los siguientes métodos, que están disponibles en todos los objetos.

Método	Descripción
Object clone( )	Crea un nuevo objeto, igual al que se duplica.
boolean equals(Object objeto)	Determina si un objeto es igual a otro.
void finalize( )	Se ejecuta antes de que un objeto no utilizado sea reciclado.
Class getClass( )	Obtiene la clase de un objeto en tiempo de ejecución.
int hashCode( )	Devuelve el código (hashcode) asociado con el objeto llamante.
void notify( )	Reanuda la ejecución de un hilo en espera en el objeto llamante.
void notifyAll( )	Reanuda la ejecución de todos los hilos en espera en el objeto llamante.
String toString( )	Devuelve una cadena que describe el objeto.
void wait( ) void wait(long *milisegundos*) void wait(long *milisegundos*, int. *nanosegundos*)	Espera a otro hilo de ejecución.

Los métodos **getClass( )**, **notify( )**, **notify All( )** y **wait( )** están declarados como **final**. Se pueden sobrescribir todos los demás métodos. Estos métodos se describen a lo largo de este libro. Sin embargo, haremos en este punto una breve descripción de dos de ellos, **equals( )** y

toString( ). El método **equals( )** compara el contenido de dos objetos. Devuelve el valor **true** si los objetos son equivalentes, y el valor **false** en caso contrario. La definición precisa de igualdad puede variar, dependiendo del tipo de objetos que han sido comparados. El método **toString( )** devuelve una cadena que contiene una descripción del objeto en el que se produce la llamada. Se llama automáticamente a este métodos cuando un objeto se va a imprimir mediante **println( )**. Muchas clases sobrescriben este método y esto les permite realizar una descripción específica de los tipos de objeto que pueden crear.

9 CAPÍTULO

Paquetes e interfaces

En este capítulo se estudian dos de las características más innovadoras de Java: los paquetes y las interfaces. Los *paquetes* son contenedores de clases que permiten mantener el espacio de nombres de clase dividido en compartimentos. Por ejemplo, un paquete nos permite crear una clase denominada **Lista**, que podemos almacenar en nuestro propio paquete sin preocuparnos de que colisione con alguna otra clase denominada **Lista** almacenada en alguna otra parte. Los paquetes son almacenados de manera jerárquica y explícitamente importados en las nuevas definiciones de clases.

En los capítulos anteriores hemos visto cómo los métodos definen la **interfaz** de los datos en una clase. Java nos permite hacer abstracción de la **interfaz** respecto de su implementación, por medio de la palabra clave **interface**. Utilizando la palabra clave **interface** se puede especificar un conjunto de métodos que pueden ser implementados por una o más clases. La palabra clave interface por sí misma no define realmente ninguna implementación. Aunque las interfaces son semejantes a las clases abstractas, tienen una característica adicional: una clase puede implementar más de una interfaz. Por el contrario, una clase sólo puede heredar de una superclase (sea abstracta o no).

Paquetes

En los ejemplos presentados en los capítulos anteriores, el nombre de cada una de las clases se tomó del mismo espacio de nombres. Esto significa que para evitar colisión de nombres cada clase debió tener un nombre distinto. Después de un tiempo, si no se ha gestionado el espacio de nombres adecuadamente, puede ocurrir que nos quedemos sin los nombres más convenientes desde un punto de vista descriptivo para cada clase individual. También hay que asegurarse de que el nombre elegido para una clase es razonablemente único y no colisiona con los nombres elegidos por otros programadores. (Imagine un pequeño grupo de programadores discutiendo para ver quién utiliza el nombre "Foobar" como nombre de clase, o imagine toda la comunidad de Internet discutiendo sobre quién utilizó por primera vez el nombre "Espresso"). Afortunadamente, Java proporciona un mecanismo para particionar el espacio de nombres de clase en partes más manejables. Este mecanismo es el paquete. El paquete es, simultáneamente, un mecanismo de control para nombres y para la visibilidad de las clases. Es posible definir clases dentro de un paquete que no sean accesibles desde un código que esté fuera de ese paquete. También es posible definir miembros de clase a los

que sólo pueden acceder miembros del mismo paquete. Esto permite que las clases de un paquete tengan un conocimiento mutuo entre ellas, que no tendrá el resto del mundo.

Definición de paquete

Es muy sencillo crear un paquete: simplemente hay que incluir el comando **package** como primera sentencia del archivo fuente de Java. Cualquier clase que se declare en ese archivo pertenecerá al paquete especificado. La sentencia **package** define un espacio de nombres en el que se almacenan las clases. Si se omite la sentencia **package**, los nombres de las clases se colocan en el paquete por omisión que no tiene nombre, por ello no nos hemos preocupado de los paquetes hasta el momento. El paquete por omisión es adecuado para pequeños ejemplos de programas, pero no sirve para aplicaciones reales. En la práctica, la mayoría de las veces será importante definir un **paquete** para nuestro código.

La forma general de la sentencia **package** es:

```
package pkg;
```

donde *pkg* es el nombre del paquete. Por ejemplo, la siguiente sentencia crea un paquete llamado **MiPaquete**.

```
Package MiPaquete;
```

Java utiliza los directorios del sistema de archivos para almacenar los paquetes. Por ejemplo, los archivos **.class** para cualquier clase que se declare como parte de **MiPaquete** se deben almacenar en un directorio llamado **MiPaquete**. Recuerde que Java distingue entre mayúsculas y minúsculas y que el nombre del directorio debe coincidir exactamente con el nombre del paquete.

Una sentencia **package** se puede incluir en más de un archivo fuente, ya que esta sentencia simplemente especifica a qué paquete pertenecen las clases definidas en un archivo, y no excluye que clases contenidas en otro archivo formen parte de ese mismo paquete. En la mayoría de los programas reales, los paquetes están conformados por muchos archivos.

Se puede crear una jerarquía de paquetes. Para ello, simplemente se separa el nombre de cada paquete del inmediatamente superior por medio de un punto. La forma general de una sentencia package de varios niveles es la siguiente:

package *pkg1*[.*pkg2*[.*pkg3*]];

Una jerarquía de paquetes debe reflejarse en el sistema de archivos del sistema. Por ejemplo, un paquete declarado como

```
package java.awt.image;
```

debe almacenarse en **java\awt\image** en el sistema de archivos de Windows. Debemos elegir los nombres de nuestros paquetes cuidadosamente, ya que no es posible cambiar de nombre a un paquete sin cambiar de nombre al directorio en el que se han almacenado las clases.

Localización de paquetes y CLASSPATH

Como se explicó antes, los paquetes se ven reflejados en directorios. Esto genera una importante pregunta: ¿Cómo sabe la máquina virtual de Java dónde buscar los paquetes que hemos creado? La respuesta a esta pregunta tiene tres partes. Primero, por omisión, la máquina virtual de Java utiliza el directorio de trabajo actual como su punto de partida. Esto es, si nuestro paquete está

en un subdirectorio del directorio actual, éste será encontrado. Segundo, podemos especificar la ruta o rutas de directorios, en donde buscar nuestros paquetes, utilizando la variable de ambiente **CLASSPATH**.

Tercero, podemos utilizar la opción –**classpath** con **java** y **javac** para especificar la ruta del directorio donde se localizan nuestras clases.

Por ejemplo, observe la siguiente especificación:

```
package MiPaquete
```

Para que un programa pueda encontrar el paquete **MiPaquete** debe ocurrir una de las tres situaciones siguientes: el programa debe ser ejecutado desde un directorio inmediatamente superior al directorio **MiPaquete**, o bien, la variable de ambiente **CLASSPATH** debe ser creada y contener la ruta del directorio **MiPaquete**, o como tercera opción, debe proporcionarse la ubicación del directorio **MiPaquete** cuando el programa sea ejecutado con el comando **java** mediante la opción –**classpath**.

Cuando las dos últimas opciones son utilizadas, la ruta del directorio **MiPaquete** no incluye el nombre del directorio **MiPaquete**. La ruta sólo específica los directorios que contienen al directorio **MiPaquete**. Por ejemplo, en Windows si la ubicación del directorio fuera

```
C:\MisProgramas\Java\MiPaquete
```

Entonces el valor para el CLASSPATH sería

```
C:\MisProgramas\Java
```

La forma más sencilla de trabajar con los ejemplos de este libro es simplemente crear los directorios de los paquetes en el directorio de trabajo y colocar los archivos **.class** en los directorios apropiados para luego ejecutar los programas desde el directorio de trabajo. Esta es la opción utilizada en el siguiente ejemplo.

Ejemplo de un paquete

Tomando en cuenta la discusión anterior, podemos probar este paquete sencillo:

```
// Un paquete sencillo
package Mipaquete;

class Balance {
  String nombre;
  double bal;

  Balance(String n, double b) {
    nombre = n;
    bal = b;
  }

  void show() {
    if (bal<0)
      System.out.print("--> ");
    System.out.println(nombre + ": $" + bal);
  }
}
class CuentaBalance {
  public static void main(String args[]) {
    Balance actual[] = new Balance[3];
```

```
    actual [0] = new Balance("K. J. Fielding", 123.23);
    actual [1] = new Balance("Will Tell", 157.02);
    actual [2] = new Balance("Tom Jackson", -12.33);

    for(int i=0; i<3; i++) actual[i].show();
  }
}
```

Llamemos a este archivo **CuentaBalance.java**, y se coloquémoslo en un directorio denominado **MiPaquete**.

Luego compilemos este archivo, asegurando que el archivo resultante **.class** también esté en el directorio **MiPaquete**. Ahora es posible la ejecución de la clase **CuentaBalance**, usando la siguiente línea de comando:

```
java MiPaquete.CuentaBalance
```

Recuerde que deberá estar un directorio arriba de **MiPaquete** cuando ejecute este comando (o bien utilizar una de las dos opciones alternativas descritas en la sección anterior para asignar a la variable de entorno CLASSPATH la ubicación del **MiPaquete**).

Tal y como se ha explicado, **CuentaBalance** es parte del paquete **MiPaquete**. Esto significa que no se puede ejecutar por sí misma, es decir, no es posible utilizar la siguiente línea de comandos:

```
java CuentaBalance
```

CuentaBalance debe estar precedida por el nombre de su paquete.

Protección de acceso

En los capítulos precedentes hemos presentado varios aspectos del mecanismo de control de acceso en Java, así como sus especificadores de acceso. Por ejemplo, ya sabemos que el acceso a los miembros **privados** de una clase sólo está permitido para otros miembros de esa clase. Los paquetes añaden otra dimensión al control de acceso. Java proporciona muchos niveles de protección que permiten un control adecuado de la visibilidad de las variables y métodos dentro de las clases, subclases y paquetes.

Las clases y los paquetes son medios que permiten la encapsulación y definen el espacio de nombres y campo de acción de las variables y los métodos. Los paquetes actúan como contenedores para las clases y otros paquetes subordinados. Las clases actúan como contenedores de datos y código. La clase es la unidad más pequeña de abstracción en Java. Dada la interacción entre clases y paquetes, Java establece cuatro categorías de visibilidad para los miembros de la clase:

- Subclases en el mismo paquete
- No subclases en el mismo paquete
- Subclases en diferentes paquetes
- Clases que no están en el mismo paquete ni son subclases

Los tres especificadores de acceso, **private**, **public** y **protected**, proporcionan diferentes formas de producir los diferentes niveles de acceso requeridos por estas categorías. La Tabla 9.1 resume las interacciones.

TABLA 9.1
Acceso a los miembros de una clase

	Privado	Sin Modificar	Protegido	Público
Misma clase	Sí	Sí	Si	Sí
Subclase del mismo paquete	No	Sí	Sí	Sí
No subclase del mismo paquete	No	Sí	Sí	Sí
Subclase de diferente paquete	No	No	Sí	Sí
No subclase de diferente paquete	No	No	No	Sí

Aunque el mecanismo de control de acceso de Java puede parecer muy complicado, es posible simplificarlo como se explica a continuación. Se puede acceder a cualquier elemento declarado como **public** desde cualquier parte del programa. No se puede acceder a un elemento declarado como **private** desde fuera de su clase. Cuando un elemento no tiene una especificación de acceso explícita, es visible para las subclases así como para otras clases que estén dentro del mismo paquete. En esto consiste el acceso por omisión. Si se desea que un elemento sea visible desde fuera del paquete actual, pero solamente para subclases derivadas directamente de la clase a que pertenece el elemento, hay que declarar al elemento como **protected**.

La Tabla 9.1 se aplica sólo a los miembros de las clases. Una clase tiene solamente dos niveles posibles de acceso: por defecto y público. Cuando se declara una clase como public, es accesible por cualquier otra parte del código. Si una clase tiene acceso por defecto, entonces sólo se puede acceder a ella por código que esté dentro del mismo paquete. Cuando una clase es **public**, ésta debe ser la única clase pública en el archivo en el cual está declarada, y el archivo debe tener el mismo nombre que esa clase pública.

Ejemplo de acceso

El siguiente ejemplo muestra todas las combinaciones de modificadores de control de acceso. En el ejemplo hay dos paquetes y cinco clases. Recuerde que las clases de los dos paquetes deben ser almacenadas en directorios que tengan el nombre de sus respectivos paquetes; en este caso, **p1** y **p2**.

El código fuente del primer paquete define tres clases: **Proteccion**, **Derivada**, y **MismoPaquete**. La primera clase define cuatro variables del tipo **int** en cada uno de los modos de protección permitidos. La variable **n** se declara con la protección por omisión, mientras que **n_pri** es **private**, **n_pro** es **protected**, y **n_pub** es **public**.

Cada una de las otras clases de este ejemplo intenta acceder a las variables en una instancia de esta primera clase. Las líneas que no se compilan debido a las restricciones de acceso, se

marcan como comentario en la línea correspondiente. Antes de cada una de estas líneas hay un comentario que indica los lugares desde los que el acceso estaría permitido.

La segunda clase, **Derivada**, es una subclase de **Proteccion** dentro del mismo paquete, **pl**. Esto garantiza que **Derivada** accede a las variables de **Proteccion** excepto a **n_pri**, la variable declarada como **private**. La tercera clase, **MismoPaquete**, no es una subclase de **Proteccion**, pero está en el mismo paquete y tiene acceso a todo excepto a **n_pri**.

Este es el archivo **Proteccion.java:**

```
package p1;

public class Proteccion {
  int n = 1;
  private int n_pri = 2;
  protected int n_pro = 3;
  public int n_pub =4;

  public Proteccion() {
    System.out.println("Constructor base");
    System.out.println("n = " + n);
    System.out.println("n_pri = " + n_pri);
    System.out.println("n_pro = " + n_pro);
    System.out.println("n_pub = " + n_pub);
  }
}
```

Este es el archivo **Derivada.java:**

```
package p1;

class Derivada extends Proteccion {
  Derivada () {
    System.out.println("Constructor de la clase Derivada");
    System.out.println("n = " + n);

//  Sólo para su clase
//  System.out.println("n_pri =" + n_pri);

    System.out.println("n_pro = " + n_pro);
    System.out.println("n_pub = " + n_pub);
  }
}
```

Este es el archivo **MismoPaquete.java:**

```
package p1;

class MismoPaquete {
  MismoPaquete () {

    Proteccion p = new Proteccion();
    System.out.println("Constructor de la clase MismoPaquete");
    System.out.println("n = " + p.n);

//  Sólo para su clase
//  System.out.println("n_pri = " + p.n_pri);
    System.out.println("n_pro = " + p.n_pro);
```

```
    System.out.println("n_pub = " + p.n_pub);
  }
}
```

A continuación se presenta el código fuente del otro paquete, **p2**. Las dos clases definidas en **p2** muestran las otras dos condiciones afectadas por el control de acceso. La primera clase, **Proteccion2**, es una subclase de **pl.Proteccion**. Esto garantiza el acceso a todas las variables de **p1.Proteccion**, excepto a **n_pri**, que se ha declarado como **private**, y a **n**, la variable declarada con la protección por omisión, es decir, a la que sólo tendrán acceso desde elementos de su clase o de su paquete, pero no desde subclases pertenecientes a otros paquetes. Finalmente, la clase **OtrosPaquetes** tiene acceso solamente a una variable, **n_pub**, la cual fue declarada como **public**.

Éste es el archivo **Proteccion2.java**:

```
package p2;

class Proteccion2 extends p1.Proteccion {
  Proteccion2() {
    System.out.println("Constructor de clase con herencia
    en paquetes distintos");

//  Sólo para su clase o paquete
//  System.out.println("n = " + n);

//  Sólo para su clase
//  System.out.println("n _pri = " + n_pri);

    System.out.println("n_pro = " + n_pro);
    System.out.println("n_pub = " + n_pub);
  }
}
```

Este es el archivo **OtroPaquete.java:**

```
package p2;

c1ass OtroPaquete {
  OtroPaquete () {
    p1.Proteccion p = new p1.Proteccion();
    System.out.println("Constructor de la clase OtroPaquete");

//  Sólo para su clase o paquete
//  System.out.println("n = " + p.n);

//  Sólo para su clase
//  System.out.println("n_pri = " + p.n_pri);

//  Sólo para su clase, subc1ase o paquete
//  System.out.println("n_ro = "+ p.n_pro);

    System.out.println("n_pub = " + p.n_pub);
  }
}
```

Para probar estos dos paquetes se pueden utilizar los dos archivos de prueba que se muestran a continuación. En primer lugar, el correspondiente al paquete **pl:**

```
// Demo paquete p1.
package p1;

// Crea instancias de las distintas clases del paquete p1.
public class Demo {
  public static void main(String args[]){
    Proteccion ob1 = new Proteccion();
    Derivada ob2 = new Derivada ();
    MismoPaquete ob3 = new MismoPaquete ();
  }
}
```

El archivo de prueba para **p2** es el siguiente:

```
// Ejemplo del paquete p2.
package p2;

// Crea instancias de las distintas clases del paquete p2.
public class Demo {
  public static void main(String args[]} {
    Proteccion2 ob1 = new Proteccion2();
    OtroPaquete ob2 = new OtroPaquete ();
  }
}
```

Importar paquetes

Después de haber visto la existencia de los paquetes, que constituyen un mecanismo adecuado para separar en compartimentos unas clases de otras, resulta sencillo entender por qué todas las clases incorporadas a Java están almacenadas en paquetes. No existen clases del núcleo de Java en el paquete sin nombre utilizado por omisión; todas las clases estándares están almacenadas en algún paquete con nombre propio. Ya que las clases contenidas en un paquete deben ser accedidas utilizando el nombre o nombres del paquete que las contiene, puede resultar tedioso escribir el nombre completo para cada clase que se quiera usar. Por este motivo, Java incluye la sentencia **import**, que permite hacer visibles ciertas clases o paquetes completos. Una vez que se ha importado una clase, se puede hacer referencia a ella usando sólo su nombre. La sentencia **import** es una comodidad para el programador, y no es técnicamente necesaria para escribir un programa. Sin embargo, al escribir un programa en el que haya una cantidad considerable de clases, la sentencia **import** permitirá ahorrar escritura.

En un archivo fuente Java, las sentencias **import** tienen lugar inmediatamente después de la sentencia **package**, en caso de que exista, y antes de la definición de cualquier clase. La forma general de la sentencia **import** es la siguiente:

import pkg1 [.pkg2] .(*nombre_de_clase* | *);

Aquí, *pkgl* es el nombre del paquete de nivel superior, y *pkg2* es el nombre del paquete subordinado contenido en el paquete exterior y separado por un punto (.). En la práctica, no existe límite para la profundidad de una jerarquía de paquetes, excepto el impuesto por el sistema de archivos. Finalmente, se puede especificar explícitamente un *nombre_de_clase* o un asterisco (*), para indicar al compilador de Java que debe importar el paquete completo. El siguiente fragmento de código muestra el uso de ambas formas:

```
import java.util.Date;
import java.io.*;
```

PRECAUCIÓN *La opción que utiliza el asterisco puede incrementar el tiempo de compilación, especialmente si se importan varios paquetes grandes. Por esta razón conviene nombrar explícitamente las clases que se quiere usar, en lugar de importar el paquete completo. Sin embargo, la utilización del asterisco no tiene efecto alguno sobre el tiempo de ejecución o tamaño de las clases.*

Todas las clases estándares de Java están almacenadas en un paquete denominado **java**. Las funciones que constituyen el lenguaje básico de Java están almacenadas en un paquete contenido en el paquete **java** y que se llama **java.lang.** Normalmente, es necesario importar cada paquete o clase que se quiera utilizar, pero debido a que Java no tiene utilidad sin la funcionalidad que se encuentra en **java.lang**, esta importación la realiza el compilador para todos los programas implícitamente. Esto es equivalente a tener al comienzo de todos los programas la siguiente línea:

```
import java.lang.*;
```

Si en dos paquetes distintos, que se importan utilizando la opción de asterisco, existen clases con el mismo nombre, el compilador no dará ningún mensaje a menos que se trate de utilizar una de esas clases. En ese caso, se obtendrá un error en tiempo de compilación y será necesario poner explícitamente el nombre de la clase especificando su paquete.

La sentencia **import** es opcional. En cualquier ubicación en que se utilice el nombre de una clase, se puede poner el *nombre completo*, que incluye los nombres de su jerarquía de paquetes completa. Por ejemplo, el siguiente fragmento de código utiliza una sentencia de importación.

```
import java.util.*;
class MiFecha extends Date {
}
```

El mismo ejemplo sin la sentencia **import** se ve así:

```
class MiFecha extends java.util.Date {
}
```

En esta segunda versión se utiliza el nombre completo de la clase **Date**.

Tal y como se muestra en la Tabla 9.1, cuando se importa un paquete, sólo aquellos elementos del paquete declarados como **public** estarán disponibles para las clases que no son subclases del código importado. Por ejemplo, si se quiere que la clase **Balance** del paquete **MiPaquete**, mostrada anteriormente, sea la única clase accesible para uso general fuera de **MiPaquete**, entonces será necesario declararla como **public** y ponerla en su propio archivo, tal y como se muestra a continuación:

```
package MiPaquete;

/* Ahora, la clase Balance, su constructor, y su método
   show() son públicos. Esto significa que pueden ser utilizados
   por código que no sea una subclase y esté fuera de su paquete.
*/
public class Balance {
```

```
  String nombre;
  double bal;

  public Balance(String n, double b) {
    nombre = n;
    bal = b;
  }

  public void show() {
    if (bal<0)
      System.out.print(" - - > ");
    System.out.println(nombre + ": $" + bal);
  }
}
```

Ahora la clase **Balance** es **public**. También su constructor y su método **show( )** son **public**. Esto significa que se puede acceder a ellos desde cualquier tipo de código que esté fuera de **MiPaquete**. Por ejemplo, en las líneas que siguen, **TestBalance** importa **MiPaquete** y entonces puede hacer uso de la clase Balance:

```
import MiPaquete.*;

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

    /*  Como Balance es pública, se puede usar la clase Balance
        y llamar a su constructor. */
    Balance test = new Balance("J. J. Jaspers", 99.88);

    test.show(); // también se puede llamar a show()
  }
}
```

Si se elimina el especificador **public** de la clase **Balance** y se intenta compilar **TestBalance**, se obtendrán los errores que se han comentado anteriormente.

Interfaces

Mediante la palabra clave **interfaz**, se puede abstraer completamente la interfaz de una clase de su implementación, es decir, usando una **interfaz** es posible especificar lo que una clase debe hacer, pero no cómo debe hacerlo. Las interfaces son sintácticamente semejantes a las clases, pero carecen de las variables de instancia, y sus métodos se declaran sin un cuerpo. En la práctica, esto implica que se pueden definir interfaces que no hagan suposiciones sobre cómo se implementan. Una vez definida una **interfaz**, cualquier número de clases puede implementarla. Además una clase puede implementar cualquier número de interfaces.

Para implementar una interfaz, una clase debe crear el conjunto completo de métodos definidos por la interfaz. Sin embargo, cada clase tiene la libertad de determinar los detalles de su implementación. Mediante la palabra clave **interfaz**, Java permite aplicar por completo la idea "una interfaz, múltiples métodos" definida por el polimorfismo.

Las interfaces se diseñan para dar soporte a la resolución dinámica de métodos durante la ejecución. Normalmente, para que un método de una clase pueda ser llamado desde otra clase, es preciso que ambas clases estén presentes durante la compilación, con el fin de que el compilador de Java pueda comprobar que el formato de los métodos es compatible.

Este requisito da lugar por sí mismo a un entorno de clases estático y no extensible. Inevitablemente, en un sistema como este, la funcionalidad aumenta a medida que se sube en la jerarquía de las clases, de forma que los mecanismos estarán disponibles para más y más subclases. Las interfaces se diseñan para evitar este problema, desconectando la definición de un método o de un conjunto de métodos de la jerarquía de herencia. Como las interfaces tienen una jerarquía distinta de la de las clases, es posible que clases que no están relacionadas en términos de jerarquía implementen la misma interfaz, lo que muestra el verdadero poder de las interfaces.

NOTA *Las interfaces aportan la mayor parte de la funcionalidad que se requiere en muchas aplicaciones. En otros lenguajes como C++ esto se consigue recurriendo a la herencia múltiple.*

Definición de una interfaz

Una interfaz se define de manera muy similar a como lo sería una clase. La forma general de definir una interfaz es la siguiente:

```
acceso nombre_interfaz {
   tipo_devuelto método1 (lista_de_parámetros);
   tipo_devuelto método2 (lista_de_parámetros);
   tipo var_final1 = valor;
   tipo var_final2 = valor;
   // ...
   tipo_devuelto métodoN (lista_de_parámetros);
   tipo var_finalN = valor;
}
```

Cuando no se indica ningún especificador de accesso, se aplica el valor de acceso por omisión y la interfaz está disponible sólo para otros miembros del paquete en que se declara. Cuando se declara como public, la interfaz puede ser utilizada por cualquier otro código. En este caso, la **interfaz** debe ser la única declarada pública en su archivo y el archivo debe tener el mismo nombre que la interfaz. *nombre* es el nombre de la interfaz y puede ser cualquier identificador válido. Observe que los métodos que se declaran no tienen cuerpo y terminan con un punto y coma después de la lista de parámetros. Son esencialmente métodos abstractos, ya que no puede haber implementación por defecto de un método declarado dentro de una interfaz. Cada clase que incluya una interfaz debe implementar todos sus métodos.

Es posible declarar variables dentro de las declaraciones de interfaces. Las variables declaradas dentro de una interfaz son implícitamente, **final** y **static**, esto significa que no pueden ser alteradas por la implementación de la clase y que deben ser inicializadas con un valor constante. Todos los métodos y variables son implícitamente **public**.

A continuación se muestra un ejemplo de definición de una interfaz sencilla, que contiene un método llamado **callback( )** que toma un sólo parámetro entero.

```
interface Callback {
  void callback(int param);
}
```

Implementación de interfaces

Una vez definida una **interfaz**, una o más clases pueden implementarla. Implementar una interfaz consiste en incluir la sentencia **implements** en la definición de la clase y crear los métodos definidos por la interfaz. La forma general de una clase que incluye la sentencia **implements** es la siguiente:

```
class nombre_de_clase [extends superclase] [implements interfaz [,interfaz...]] {
  // cuerpo de la clase
}
```

Si una clase implementa más de una interfaz, las interfaces se separan con comas. Si una clase implementa dos interfaces que declaran al mismo método, entonces los clientes de ambas interfaces deberán usar al mismo método. Los métodos que implementan una interfaz deben declararse como **public**. Además, la firma del método implementado debe coincidir exactamente con el formato especificado en la definición de la **interfaz**.

El siguiente ejemplo muestra una clase que implementa la interfaz **Callback**, presentada anteriormente.

```
class Cliente implements Callback {
  // Implementa la interfaz Callback
  public void callback(int p) {

    System.out.println("callback llamado con" + p);
  }
}
```

Observe que se declara **callback( )** usando el especificador de acceso public.

NOTA *Cuando se implementa un método de una interfaz, debe ser declarado como* ***public****.*

Se permite y es común que las clases que implementan interfaces definan miembros adicionales propios. Por ejemplo, la siguiente versión de **Cliente** implementa **callback( )** y añade el método **nonIfaceMeth( ):**

```
class Cliente implements Callback {
  // Implementa la interfaz Callback
  public void callback(int p) {
    System.out.println("callback llamado con" + p);
  }

  void nonIfaceMeth() {
    System.out.println("Las clases que implementan interfaces " +
                       "pueden definir también otros miembros.");
  }
}
```

Acceso a la clase implementada mediante referencias del tipo de la interfaz

Se pueden declarar variables como referencia a objetos que usan una interfaz como tipo en lugar de una clase. Se puede hacer referencia a cualquier instancia de cualquier clase que implementa una interfaz declarada por medio de tales variables. Cuando se llama a un método por medio de una de estas referencias, se llamará a la versión correcta que se basa en la

instancia actual de la interfaz que está siendo referenciada. Ésta es una de las características clave de las interfaces. El método que se va a ejecutar se determina dinámicamente durante la ejecución, permitiéndose que las clases en las que se encuentra dicho método se creen después del código llamante, que puede seleccionar una interfaz sin tener ningún conocimiento sobre el método"llamado". Este proceso es similar al que se tenía al utilizar una referencia de una superclase para acceder a un objeto de una subclase, tal y como se describió en el Capítulo 8.

PRECAUCIÓN *Teniendo en cuenta que la búsqueda dinámica de un método durante la ejecución supone un mayor tiempo de proceso, en comparación con la llamada normal a un método en Java, conviene ser cuidadosos y no utilizar interfaces innecesariamente en códigos cuyo rendimiento es crítico.*

En el siguiente ejemplo se llama al método **callback( )** por medio de una variable de referencia a la interfaz:

```
class TestIface {
  public static void main(String args[]) {
    Callback c = new Cliente();
    c. callback (42) ;
  }
}
```

La salida de este programa es la siguiente:

```
callback llamado con 42
```

Observe que se ha declarado la variable **c** del tipo de la interfaz **Callback**, aunque se le ha asignado una instancia de **Cliente**. Aunque se puede utilizar **c** para acceder al método **callback( )**, no sirve para acceder a otros miembros de la clase **Cliente**. Una variable de referencia a una interfaz sólo tiene conocimiento de los métodos que figuran en la declaración de su **interfaz**. Por lo tanto, **c** no podría utilizarse para acceder al método **nonIfaceMeth( )**, ya que éste está definido en **Cliente** pero no en **Callback**.

El ejemplo anterior muestra, de una manera mecánica, cómo una variable de referencia a una interfaz puede acceder a la implementación de un objeto, pero no muestra el poder del polimorfismo de tal referencia. Como ejemplo de esta utilidad, se crea, en primer lugar, una segunda implementación de **Callback**:

```
// Otra implementación de Callback.
class OtroCliente implements Callback {
  // Implementa la interfaz de Callback
  public void callback(int p) (
    System.out.println("Otra versión de callback");
    System.out.println("El cuadrado de p es . + (p*p));
  }
}
```

Ahora probemos la siguiente clase:

```
class TestIface2 {
public static void main(String args[]) {
```

```
  Callback c = new Cliente();
    OtroCliente ob = new OtroCliente();

    c.callback(42);

    c = ob; // c ahora es una referencia a un objeto de la clase OtroCliente
    c.callback(42);
  }
}
```

La salida de este programa es la siguiente:

```
    callback llamado con 42
    Otra versión de callback
    El cuadrado de p es 1764
```

Como se puede ver, la versión del método **callback( )** llamada se determina por el tipo del objeto al que hace referencia **c** en tiempo de ejecución. Aunque éste es un ejemplo muy sencillo, enseguida se verá otro más práctico.

Implementaciones parciales

Cuando una clase incluye una interfaz, pero no implementa completamente los métodos definidos por esa interfaz, entonces la clase debe ser declarada como **abstracta**, utilizando la palabra clave abstract. Por ejemplo:

```
abstract class Incomplete implements Callback {
  int a, b;
  void show() {
    System.out.println(a + " " + b);
  }
  // ...
}
```

Donde **Incomplete** una clase que no implementa el método **callback( )** y debe ser declarada abstracta. Cualquier clase que herede **Incomplete** debe implementar el método **callback( )**, o bien ser declarada también como **abstracta**.

Interfaces anidadas

Una interfaz puede ser declarada como miembro de una clase o de otra interfaz, cuando ello ocurre la interfaz es llamada una *interfaz miembro* o una *interfaz anidada*. Una interfaz anidada puede ser declarada como **public**, **private** o **protected**. Esto es diferente a lo que sucede con las interfaces no anidadas las cuales deben ser declaradas como **public** o con el nivel de acceso por omisión, como se describió antes. Cuando una interfaz anidada es utilizada fuera de su ámbito, ésta debe ser llamada con su nombre y el de la clase o interfaz en la cual está contenida. De manera que, fuera de la clase o interfaz en la cual la interfaz anidada se encuentra el nombre a utilizar debe ser especificado completamente.

A continuación un ejemplo que muestra el uso de interfaces anidadas:

```
// Ejemplo de interfaces anidadas

// Esta clase contiene una interface anidada
class A {

  // ésta es la interfaz anidada
  public interface NestedIF {
```

```
    boolean isNotNegative(int x);
  }
}

// B implementa la interfaz anidada
class B implements A.NestedIF {
  public boolean isNotNegative(int x) {
    return x < 0 ? false : true;
  }
}

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

    // Utiliza una referencia a una interfaz anidada
    A.NestedIF nif = new B();

    if(nif.isNotNegative(l0))
      System.out.println("10 es un número no negativo");
    if(nif.isNotNegative(-12))
      System.out.println("Esto no aparecerá en pantalla");
  }
}
```

Observe que **A** define una interfaz miembro llamada **NestedIF** la cual es declarada **public**. Enseguida **B** implementa la interfaz anidada a través de

```
implements A.NestedIF
```

Observe también que el nombre utilizado para implementar **NestedIF** es la especificación completa que incluye nombre de la interfaz anidada y de su clase contenedora. Dentro del método **main** se crea una referencia a **A.NestedIF** llamada **nif** y se le asigna una referencia a un objeto de clase **B**. Esto es correcto debido a que **B** implementa a **A.NestedIF.**

Utilizando interfaces

Veamos un caso más práctico que nos ayude a entender la potencia de las interfaces. En los capítulos anteriores se desarrolló una clase denominada **Stack** que implementaba una pila sencilla de tamaño fijo. Sin embargo, existen otras formas de implementar una pila. Por ejemplo, la pila puede ser de tamaño fijo o variable. Además, la pila se puede almacenar en un arreglo, una lista, un árbol binario, etc. Independientemente de cómo se haya implementado la pila, la interfaz es siempre la misma, es decir, los métodos **push( )** y **pop( )** definen la interfaz de la pila al margen de los detalles de la implementación. Como la interfaz de la pila es independiente de su implementación, es fácil definir dicha interfaz, dejando para cada implementación los detalles más específicos. Veamos dos ejemplos.

En primer lugar se presenta la interfaz que define una pila de enteros. Coloquemos este código en un archivo denominado **IntStack.java.** De está interfaz construiremos dos implementaciones más adelante.

```
// Definición de la interfaz de una pila de enteros.
interfaz IntStack {
  void push(int item); // almacena un elemento
  int pop(); // recupera un elemento
}
```

El siguiente programa crea una clase llamada **FixedStack** que implementa una versión de una pila de enteros de longitud fija.

```
// Esta implementación de IntStack utiliza almacenamiento fijo.
class FixedStack implements IntStack {
  private int stck[];
  private int tos;

  // Reserva espacio e inicializa la pila
  FixedStack(int size) {
    stck = new int[size];
    tos = -1;
  }

  // Coloca un elemento en la pila
  public void push(int item) {
    if(tos==stck.length-1) // se utiliza la variable miembro length
    para conocer el tamaño del arreglo
      System.out.println("La pila está llena.");
    else
      stck[++tos] = item;
  }

// Retira un elemento de la pila
public int pop() {
  if(tos < 0) {
    System.out.println("La pila está vacía .");
    return 0;
  }
  else
    return stck[tos--];
  }
}

c1ass IFTest (
  public static void main(String args[]) {
    FixedStack miPilal = new FixedStack(5);
    FixedStack miPila2 = new FixedStack(8) ;

    // Se almacenan algunos números en la pila
    for(int i=0; i<5; i++) miPilal.push(i);
    for(int i=0; i<8; i++) miPila2.push(i);

    // Se retiran esos números de la pila
    System.out.println ("Contenido de miPilal:");
    for(int i=0; i<5; i++)
      System.out.println(miPilal.pop());

    System.out.println("Contenido de miPila2:") ;
    for(int i=0; i<8; i++)
    System.out.println(miPila2.pop());
  }
}
```

A continuación se muestra otra implementación de **IntStack** que crea una pila dinámica utilizando la misma definición de la **interfaz**. En esta implementación cada pila se construye con

una longitud inicial. Cuando se excede esta longitud la pila se incrementa de tamaño. Cada vez que se necesita más espacio se duplica el tamaño de la pila.

```
// Implementación de una pila de tamaño "creciente".
class DynStack implements IntStack {
  private int stck[];
  private int tos;

  // Se reserva espacio y se inicializa la pila
  DynStack(int size) {
    stck = new int[size];
    tos = -1;
  }

  // Se almacena un elemento en la pila
  public void push(int item) {
    // Si la pila está llena, se reserva espacio para una pila mayor
    if(tos==stck.length-1) {
      int temp[] = new int[stck.length * 2]: // Se duplica el tamaño
      for(int i=0; i<stck.length; i++) temp[i] = stck[i];
      stck = temp:
      stck[++tos] = item;
    }
    else
      stck[++tos] = item;
  }

// Se retira un elemento de la pila
public int pop() {
  if (tos < 0) (
    System.out.println("La pila está vacía."):
    return 0:
  }
  else
    return stck[tos-];
  }
}

class IFTest2 (
  public static void main(String args[]) {
    DynStack miPila1 = new DynStack(5);
    DynStack mipila2 = new DynStack(8);

    // Estos ciclos hacen que crezca el tamaño de cada pila
    for(int i=0; i<12; i++) miPila1.push(i);
    for(int i=0; i<20; i++) miPila2.push(i);

    System.out.println("Contenido de miPila1:");
    for(int i=0; i<12; i++)
      System.out.println(miPila1.pop());

    System.out.println("Contenido de miPila2:");
    for(int i=0; i<20; i++)
      System.out.println(miPila2.pop());
  }
}
```

La siguiente clase utiliza las dos implementaciones, **FixedStack** y **DynStack**, por medio de una referencia a la interfaz. Esto significa que las llamadas a los métodos **push( )** y **pop( )** se resuelven durante el tiempo de ejecución y no en el tiempo de compilación.

```
/*  Se crea una variable de tipo interfaz y
    se accede a la pila a través de ella.
*/
class IFTest3{
  public static void main(String args[]) {
    IntStack miPila; // Se crea una variable de referencia a la interfaz
    DynStack ds = new DynStack(5);
    FixedStack fs = new FixedStack(8);

    miPila = ds; // Se carga la pila dinámica
    // Se coloca algunos números en la pila
    for(int i=0; i<12; i++) miPila.push(i);

    miPila = fs; // Se carga la pila de tamaño fijo
    for(int i=0; i<8; i++) miPila.push(i);

    miPila = ds;
    System.out.println("Valores de la pila dinámica:");
    for(int i=0; i<12; i++)
      System.out.println(miPila.pop());

    miPila = fs;
    System.out.println("Valores de la pila de tamaño fijo:");
    for(int i=0; i<8; i++)
      System.out.println(miPila.pop());
  }
}
```

En este programa, **miPila** es una referencia a la interfaz **IntStack**. Por lo tanto, cuando se refiere a **ds**, utiliza las versiones de **push( )** y **pop( )** definidos por la implementación **DynStack**, mientras que cuando se refiere a **fs**, utiliza las versiones de **push( )** y **pop( )** definidas por **FixedStack**. Tal y como se ha explicado, estas determinaciones se hacen en tiempo de ejecución. El acceso a implementaciones múltiples de una interfaz por medio de una variable de referencia de la interfaz es la forma más poderosa de que dispone Java para lograr el polimorfismo en tiempo de ejecución.

Variables en interfaces

Se pueden utilizar interfaces para importar constantes compartidas por múltiples clases, declarando una interfaz que contiene las variables inicializadas con los valores deseados. Cuando se incluye esa interfaz en una clase, es decir, cuando se "implementa"la interfaz, todos esos nombres de variables estarán dentro del ámbito como constantes. Esto es similar a la utilización de un archivo cabecera en C/C++ para crear un mayor número de constantes **#defined** o declaraciones **const**. Si una interfaz no contiene métodos, entonces cualquier clase que incluya esa interfaz no necesita hacer ninguna implementación.

Es como si esa clase estuviera importando variables "constantes"en el espacio de nombres como variables **final**. El siguiente ejemplo utiliza esta técnica para implementar un mecanismo de decisión automática.

```
import java.util.Random;

interface ConstantesCompartidas {
  int NO = O;
  int YES = 1;
  int MAYBE = 2;
  int LATER = 3;
  int SOON = 4;
  int NEVER = 5;
}

class Pregunta implements ConstantesCompartidas {
  Random rand = new Random ();
  int preguntar() {
    int prob = (int) (100 * rand.nextDouble () ) ;
    if (prob < 30)
      return NO; // 30%
    else if (prob < 60)
      return YES; // 30%
    else if (prob < 75)
      return LATER; // 15%
    el se if (prob < 98)
      return SOON; // 13%

    else
      return NEVER; // 2%
  }
}

class Preguntame implements ConstantesCompartidas {
  static void respuesta(int result) {
    switch (result) {
      case NO:
        System.out.println("No");
        break;
      case YES:
        System.out.println("Si");
        break;
      case MAYBE:
        System.out.println("Puede ser");
        break;
      case LATER:
        System.out.println ("Más tarde");
        break;
      case SOON:
        System.out.println("Pronto");
        break;
      case NEVER:
        System.out.println("Nunca");
        break;
    }
  }

  public static void main(String args[]) {
    Pregunta q = new Pregunta{};
    respuesta(q.preguntar());
```

```
    respuesta (q.preguntar());
    respuesta (q.preguntar());
    respuesta (q.preguntar{));
  }
}
```

Observe que este programa utiliza una de las clases estándar de Java, la clase **Random**. Esta clase proporciona números pseudo aleatorios. Contiene varios métodos que permiten obtener números aleatorios en la forma requerida por el programa. En este ejemplo, se utiliza el método **nextDouble().** Este método devuelve números aleatorios pertenecientes al intervalo 0.0 a 1.0.

En este ejemplo, las dos clases, **Pregunta** y **Preguntame**, implementan la interfaz **ConstantesCompartidas** en la que se definen **NO**, **YES**, **MAYBE**, **SOON**, **LATER** y **NEVER**. Dentro de cada clase, el código se refiere a estas constantes como si cada clase las hubiera heredado o definido directamente. En la salida generada por este programa se puede observar que los resultados son diferentes cada vez que se ejecuta.

```
Más tarde
Pronto
No
Sí
```

Las interfaces se pueden extender

Una interfaz puede heredar otra mediante el uso de la palabra clave **extends**. La sintaxis es la misma que en el caso de la herencia de clases. Cuando una clase implementa una interfaz que hereda otra interfaz, debe proporcionar las implementaciones para todos los métodos definidos en la cadena de herencia de la interfaz. A continuación se presenta un ejemplo:

```
// Una interfaz puede extender otra.
interface A {
  void methl () ;
  void meth2 () ;
}

// B ahora incluye los métodos methl() y meth2() y añade meth3().
interface B extends A {
  void meth3();
}

// Esta clase debe implementar todos los métodos de A y B
class MiClase implements B {
  public void methl() {
    System.out.println("Implementa methl() .");
  }

  public void meth2() {
    System.out.println("Implementa meth2() .");
  }

  public void meth3() {
    System.out.println("Implementa meth3() .");
  }
```

```
}

class IFExtend {
  public static void main(String arg[]) {
    MiClase ob = new MiClase();

    ob.methl ();
    ob.meth2 ();
    ob.meth3 ();
  }
}
```

Si se intenta eliminar la implementación de **methl( )** en **MiClase**, se producirá un error de compilación. Como se ha comentado anteriormente, cada clase que implementa una interfaz debe implementar todos los métodos definidos en la interfaz, incluyendo cualquiera que se haya heredado de otras interfaces.

Aunque los ejemplos que se presentan en este libro no utilizan con frecuencia paquetes o interfaces, estas dos herramientas son una parte importante del entorno de programación de Java. En general, todos los programas reales que se escriben en Java estarán contenidos en paquetes. Además, muchos de ellos también implementarán interfaces. Por este motivo es importante estar familiarizado con su uso.

CAPÍTULO 10

Gestión de excepciones

En este capítulo se examina el mecanismo para la gestión de excepciones de Java. Una *excepción* es una condición anormal que surge en una secuencia de código en tiempo de ejecución. En otras palabras, una excepción es un error en tiempo de ejecución. En los lenguajes de programación que no disponen de gestión de excepciones, los errores deben ser revisados y gestionados manualmente, mediante el uso de códigos de error. Esta solución es tan pesada como engorrosa. La gestión de excepciones de Java evita estos problemas e incorpora el manejo de errores en tiempo de ejecución al mundo de la programación orientada a objetos.

Fundamentos de la gestión de excepciones

Una excepción, en Java, es un objeto que describe una condición excepcional, es decir un error que ha ocurrido en una parte de un código. Cuando surge una condición excepcional, se crea un objeto que representa esa excepción y se *envía* al método que ha originado el error. Ese método puede decidir entre gestionar él mismo la excepción o pasarla. En cualquiera de los dos casos, en algún punto, la excepción es *capturada* y procesada. Las excepciones pueden ser generadas por el intérprete Java o por el propio código. Las excepciones generadas por Java se refieren a errores fundamentales que violan las reglas del lenguaje Java o las restricciones del entorno de ejecución de Java. Las excepciones generadas por el código se usan normalmente para informar de alguna condición de error a un método que llamó a otro.

La gestión de excepciones en Java se lleva a cabo mediante cinco palabras clave: **try**, **catch**, **throw**, **throws** y **finally**. A continuación se describe brevemente su funcionamiento. Las sentencias del programa que se quieran monitorear, se incluyen en un bloque **try**. Si una excepción ocurre dentro del bloque **try**, ésta es lanzada. El código puede capturar esta excepción, utilizando **catch**, y gestionarla de forma racional. Las excepciones generadas por el sistema son automáticamente enviadas por el intérprete Java. Para enviar manualmente una excepción se utiliza la palabra clave **throw**. Se debe especificar mediante la cláusula **throws** cualquier excepción que se envíe desde un método al método exterior que lo llamó. Se debe poner cualquier código que el programador desee que se ejecute siempre, después de que un bloque **try** se complete, en el bloque de la sentencia **finally**.

La forma general de un bloque de gestión de excepciones es la siguiente:

```
try {
   // bloque de código a monitorear por errores
}
```

```
catch (TipoExcepcionl exOb) {
  // gestor de excepciones para ExcepciónTipol
}
catch (TipoExcepcion2 exOb) {
  // gestor de excepciones para ExcepciónTipo2
}
// ...
finally {
  // bloque de código que se debe ejecutar después de que el bloque try termine
}
```

Donde *TipoExcepcion* es el tipo de la excepción que se ha producido. El resto de este capítulo describe cómo se aplica la gestión de excepciones.

Tipos de excepciones

Todos los tipos de excepciones son subclases de la clase incorporada por Java, **Throwable**. Por ello **Throwable** se encuentra en la parte superior de la jerarquía de clases de excepción. Inmediatamente después de **Throwable** se encuentran dos subclases que dividen las excepciones en dos grupos. Un grupo es el encabezado por la clase **Exception**. Esta clase se utiliza para condiciones excepcionales que los programas deben capturar. Ésta es también la clase de la que se derivan las subclases necesarias para crear los tipos propios de excepciones. Una subclase importante de **Exception** es **RuntimeException**. Las excepciones de tipo **RuntimeException** son definidas automáticamente por los programas, e incluyen, por ejemplo, la división por cero, o la utilización de un índice de arreglo no válido.

El otro grupo está encabezado por la clase **Error**, que define excepciones no esperadas por el programa en condiciones normales. El intérprete Java utiliza las excepciones del tipo Error para indicar errores relacionados con el propio ambiente de ejecución. Un ejemplo de este tipo de error es el desbordamiento de la pila (mejor conocido por su nombre en inglés como Stack Overflow). En este capítulo no se tratarán las excepciones de este tipo, ya que se crean en respuesta a fallos catastróficos que normalmente no pueden ser gestionados por el programa.

Excepciones no capturadas

Antes de aprender cómo se manejan las excepciones en los programas, es interesante ver lo que ocurre cuando no se gestionan de ninguna forma. Este pequeño programa incluye una expresión que intencionadamente ocasiona el error debido a la división entre cero.

```
class Exc0 {
  public static void main(String args[]) {
    int d = 0;
    int a = 42 / d;
  }
}
```

Cuando el intérprete Java detecta un intento de división por cero, genera un nuevo objeto de la clase **Exception** y lanza dicha excepción. Esto da lugar a que se detenga la ejecución de **Exc0**, ya que una vez que la excepción ha sido lanzada, debe ser *capturada* por un gestor de

excepciones y tratada inmediatamente. En este ejemplo no se han proporcionado gestores de excepciones propios, de forma que la excepción es capturada por el gestor por omisión del intérprete Java. Cualquier excepción que no sea capturada por nuestro programa será finalmente procesada por el gestor por omisión, que presentará un mensaje con la descripción de la excepción, imprimirá el trazado de la pila del lugar donde se produjo la excepción y finaliza el programa.

La salida generada cuando se ejecuta este ejemplo:

```
java.lang.ArithmeticException: / by zero
        at Exc0.main(Exc0.java:4)
```

Observe cómo el nombre de la clase, **Exc0**; el nombre del método, **main**; el nombre del archivo, **Exc0.java**; y el número de la línea, **4**, están todos ellos incluidos en el trazado de la pila. Igualmente, el tipo de excepción lanzada, denominada **ArithmeticException**, es una subclase de **Exception**, cuyo nombre describe de manera más específica el tipo de error que se ha producido. Como se verá más adelante, Java incorpora varios tipos de excepciones que se ajustan a las distintas clases de errores en tiempo de ejecución que se pueden generar.

La traza de la pila siempre muestra la secuencia de llamadas a métodos en el momento del error. A continuación se presenta otra versión del ejemplo anterior que introduce el mismo error pero en un método distinto de **main( ):**

```
class Exc1 {
  static void subroutine() {
    int d = 0;
    int a = 10 / d;
  }
  public static void main(String args[]) {
    Excl.subroutine();
  }
}
```

El trazado de la pila resultante del gestor de excepciones por omisión muestra la pila de llamadas completa:

```
java.lang.ArithmeticException: / by zero
      at Excl. subroutine (Excl.java:4)
      at Excl.main(Excl.java:7)
```

Como se puede ver, el final de la pila es la línea 7 de **main**, que es donde se llama al método **subroutine( )**, el cual provocó la excepción en la línea 4. La pila de llamadas es muy útil en la depuración porque muestra exactamente la secuencia de pasos que condujo al error.

Utilizando try y catch

Aunque el sistema de gestión de excepciones que proporciona el intérprete Java es útil cuando se trata de depurar programas, normalmente el programador prefiere gestionar por sí mismo una excepción. Esto tiene dos ventajas. En primer lugar permite corregir el error. En segundo lugar, evita que el programa termine automáticamente. Muchos usuarios se confundirían, al menos, si su programa suspendiera la ejecución e imprimiese un trazado de la pila siempre que se produjera un error. Afortunadamente es bastante sencillo evitar esto.

Para protegerse de esta situación y gestionar un error en tiempo de ejecución, lo único que hay que hacer es encerrar el código que se quiera monitorear dentro de un bloque **try**. Inmediatamente después del bloque **try**, se incluye la sentencia **catch**, que especifica el tipo de excepción que se desea capturar. El siguiente programa muestra cómo se puede conseguir esto de forma sencilla, con un bloque **try** y una sentencia **catch** que procesa la excepción de tipo **ArithmeticException** generada por el error debido a la división por cero.

```
class Exc2 (
  public static void main{String args[]) {
    int d, a;

    try { // monitoreo de un bloque de código.
      d = 0;
      a = 42 / d;
      System.out.println("Esto no se imprimirá.");
      } catch (ArithmeticException e) { // captura el error
        de la división entre cero
        System.out.println("División entre cero.");
    }
    System.out.println("Después de la sentencia catch.");
  }
}
```

Este programa genera la siguiente salida:

```
División entre cero.
Después de la sentencia catch.
```

Observe que la llamada al método **println( )** dentro del bloque **try** no se ejecutará nunca. Una vez que se lanza una excepción, el control del programa se transfiere del bloque **try** al bloque **catch**. La ejecución nunca vuelve al bloque **try** desde el **catch**. Por este motivo, no se presenta en pantalla la frase "Esto no se imprimirá". Una vez ejecutada la sentencia **catch**, el control del programa continúa con la línea que sigue en el programa al conjunto **try/catch.**

Un bloque **try** y su correspondiente sentencia **catch** forman una unidad. El campo de acción de una sentencia **catch** se restringe a aquellas sentencias especificadas por la sentencia **try** que le precede inmediatamente. Una sentencia **catch** no puede capturar una excepción lanzada por otra sentencia **try**, excepto en el caso de sentencias **try** que se describe a continuación. Las sentencias protegidas por la sentencia **try** se deben encerrar entre llaves, es decir deben estar contenidas en un bloque. No se puede utilizar **try** sin las llaves.

El objetivo de la mayor parte de las sentencias **catch** bien construidas debe ser resolver una condición excepcional y continuar como si el error nunca hubiera ocurrido. Por ejemplo, en el siguiente programa cada iteración del ciclo **for** calcula dos enteros aleatorios. Esos dos enteros se dividen uno por el otro, y el cociente se utiliza como divisor del valor 12345. El resultado final se almacena en la variable **a**. Si en cualquiera de las dos operaciones se produce la división por cero, este error es capturado, el valor de **a** se pone a cero y el programa continúa.

```
// Gestión de una excepción
import java.util.Random;

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

```
    int a=0, b=0, c=0;
    Random r = new Random() ;

    for(int i=0; i<32000; i++) {
    try {
      b = r.nextInt();
      c = r.nextInt();
      a = 12345 / (b/c);
    } catch (ArithmeticException e) {
      System.out.println("División entre cero.");
      a = 0; // se asigna cero a la variable a y se continúa
    }
    System.out.println("a: " + a);
  }
 }
}
```

Descripción de una excepción

La clase **Throwable** sobrescribe el método **toString( )** definido por la clase **Object** para que devuelva una cadena que contiene la descripción de la excepción. De esta forma se puede presentar esta descripción mediante la sentencia **println( )**, simplemente pasándole la excepción como argumento. Por ejemplo, el bloque **catch** del programa anterior se puede reescribir de la siguiente forma:

```
catch (ArithmeticException e) {
  System.out.println("Excepción: " + e);
  a = 0; // se asigna cero a la variable a y se continúa
}
```

Cuando se sustituye esta versión en el programa anterior, y se ejecuta el programa, cada error de división entre cero presenta el siguiente mensaje:

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

Aunque en este contexto no tiene un gran interés, la posibilidad de presentar la descripción de una excepción sí tiene importancia en otras circunstancias, en especial cuando se están experimentando con excepciones o depurando un programa.

Cláusulas catch múltiples

En algunos casos, una misma secuencia de código puede activar más de un tipo de excepción. Para gestionar esta situación, se pueden utilizar dos o más sentencias **catch**, capturando cada una de ellas un tipo diferente de excepción. Cuando se lanza una excepción, se inspecciona por orden cada sentencia **catch**, y se ejecuta la primera cuyo tipo coincide con la excepción. Después de la ejecución de una sentencia **catch**, las demás no se ejecutan, y la ejecución continúa después del bloque **try/catch**. En el siguiente ejemplo se capturan dos tipos de excepción diferentes:

```
// Demostración de múltiples sentencias catch.
classMultiCatch {
    public static void main(String args[]){
      try {
```

```
      int a = args.length;
      System.out.println("a = " + a);
      int b = 42 / a;
      int c [] = { 1 };
      c[42] = 99;
    } catch(ArithmeticException e) {
      System.out.println("División entre 0: " + e);
    } catch(ArraylndexOutOfBoundsException e) {
      System.out.println("Índice del arreglo fuera de rango: " + e);
    }
    System.out.println("Después de los bloques try/catch.");
  }
}
```

En el programa se produce la excepción de división por cero si se ejecuta sin parámetros en la línea de comandos, ya que **a** es igual a cero. No se producirá este error si se pasa un argumento en la línea de comandos que asigne **a** a un valor mayor que cero. Sin embargo, aparecerá una excepción del tipo **ArrayIndexOutOfBoundsException**, ya que el arreglo de enteros **c** tiene una longitud igual a **l**, y el programa intenta asignar un valor a **c[42].**

A continuación se muestra la salida generada ejecutando el programa de las dos formas:

```
C:\>java MultiCatch
a = 0
División entre 0: java.lang.ArithmeticException: / by zero
Después de los bloques try/catch.

C:\>java MultiCatch TestArg
a = 1
Índice del arreglo fuera de rango:
java.lang.ArraylndexOutOfBoundsException: 42
Después de los bloques try/catch.
```

Cuando se utilizan varias sentencias **catch**, es importante recordar que las subclases de la clase Exception deben estar delante de cualquiera de sus superclases. Esto se debe a que una sentencia **catch** que utiliza una superclase captura las excepciones de sus subclases y, por lo tanto, éstas no se ejecutarán si están después de la superclase. Además, en Java se produce un error si hay código no alcanzable. Como ejemplo, consideremos el siguiente programa:

```
/*  Este programa contiene un error.

    Una subclases debe ir delante de su superclase
    en una serie sentencias catch. Si no,
    se creará código inalcanzable y eso
    resultará en un error en tiempo de compilación.
*/
class SuperSubCatch {
  public static void main(String args[]) {
    try {
      int a = 0;
      int b = 42 / a;
    } catch(Exception e) {
```

```
      System.out.println("Capturando una excepción genérica.");
    }
    /*  Este catch nunca se alcanzará porque la excepción de tipo
        ArithmeticException es una subclase de la clase Exception. */
    catch(ArithmeticException e) { // ERROR - esto no se ejecuta
      System.out.println( "Esto nunca se ejecuta.");
    }
  }
}
```

Si se intenta compilar este programa, se recibirá un mensaje de error que establece que no se accede a la segunda sentencia **catch** porque la excepción ya ha sido capturada. Como **ArithmeticException** es una subclase de la clase **Exception**, la primera sentencia **catch** gestionará todos los errores que se basan en la clase **Exception**, incluyendo **ArithmeticException**. Esto significa que la segunda sentencia **catch** no se ejecuta. Para solucionar el problema basta colocar en orden inverso a las sentencias **catch**.

Sentencias try anidadas

La sentencia **try** puede ser anidada. Esto es, una sentencia **try** puede estar dentro del bloque de otro **try**. Cada vez que una sentencia **try** es ingresada, el contexto de esa excepción se vuelve a colocar en la pila. Si una sentencia **try** colocada en el cuerpo de otra sentencia **try** no realiza la gestión de una excepción particular, la pila es liberada y la siguiente sentencia **try** inspeccionada en busca de una coincidencia. Esto continua hasta que una de las sentencias **catch** tiene éxito o hasta que todas las sentencias **try** anidadas han sido pasadas. Si ninguna sentencia **catch** coincide, la máquina virtual de Java atrapará la excepción. Veamos un ejemplo de sentencias **try** anidadas:

```
// Ejemplo de sentencias try anidadas
class NestTry {
  public static void main(String args[]) {
    try {
      int a = args.length;

      /*  Si no se utiliza ningún argumento en la línea
          de comandos, la siguiente sentencia generará una
          excepción de división entre cero. */
      int b = 42 / a;

      System.out.println("a = " + a);

      try { // bloque try anidado
        /*  Si se utiliza un argumento en la línea de órdenes
            entonces se genera una excepción de división entre cero
            en el siguiente código */
        if(a==1) a = a/(a-a); // división entre cero

        /*  Si se utilizan dos argumentos en la línea de órdenes
            entonces se genera una excepción de índice de arreglo
              fuera de rango */
        if (a==2) {
            int c [] = { 1 };
```

```
            c[42] = 99; // genera una excepción por el índice
              de arreglo fuera de rango
          }
        } catch(ArraylndexOutOfBoundsException e) {
          System.out.println("Índice del arreglo fuera de rango: " + e);
        }

      } catch(ArithmeticException e) {
        System.out.println("División entre 0: " + e);
      }
    }
}
```

Como se puede ver, este programa anida un bloque **try** con otro. El programa trabaja como sigue. Cuando se ejecuta el programa sin argumentos en la línea de órdenes, una excepción de división entre cero se genera por el bloque **try** exterior. La ejecución de programa con un argumento en la línea de órdenes genera una excepción de división entre cero en el bloque **try** anidado. Dado que el bloque interno no atrapa la excepción, ésta es enviada al bloque **try** externo, donde es gestionada. Si ejecutamos el programa con dos argumentos en la línea de órdenes, una excepción de índice de arreglo fuera de rango se genera en el bloque interno. Éste es un ejemplo de la salida desplegada por el programa anterior:

```
C:\>java NestTry
División entre 0: java.lang.ArithmeticException: / by zero

C:\>java NestTry Uno
a = 1
División entre 0: java.lang.ArithmeticException: / by zero

C:\>java NestTry Uno Dos
a = 2
Índice del arreglo fuera de rango:
  java.lang.ArrayIndexOutOfBoundsException: 42
```

El anidamiento de sentencias **try** puede ocurrir de manera menos evidente cuando están de por medio invocaciones a métodos. Por ejemplo, podemos encerrar una llamada a un método en un bloque **try** y dentro de ese método colocar otra sentencia **try**. En este caso, la sentencia **try** en el método se considera anidada dentro del bloque **try** externo que mandó llamar al método. A continuación se presenta el programa previo reorganizado para que el bloque anidado **try** ahora esté localizado dentro del método **nesttry**.

```
/*  La sentencia try puede estar anidada implícitamente
    vía llamadas a métodos */
class MethNestTry {
  static void nesttry (int a) {
    try ( // bloque try anidado
      /*  Si se utiliza un argumento en la línea de órdenes,
          se generará una excepción de división entre cero
          en el siguiente código. */
      if (a==1) a = a / (a - a); // división entre cero
    /*  Si se utilizan dos argumentos en la línea de órdenes
        entonces se genera una excepción de índice de arreglo
          fuera de rango */
```

```
    if (a==2) {
        int c [] = { 1 };
        c[42] = 99; // genera una excepción por el índice de arreglo
        fuera de rango
    }
  } catch(ArraylndexOutOfBoundsException e) {
    System.out.println("Índice del arreglo fuera de rango: " + e);
  }
}

public static void main (String args []) {
  try {
    int a = args.length;

    /*  Si no se utiliza ningún argumento en la línea
        de comandos, la siguiente sentencia generará una
        excepción de división entre cero. */
      int b = 42 / a;
      System.out.println("a = " + a);

      nesttry (a) ;
    } catch(ArithmeticException e) (
      System.out.println("División entre 0: " + e);
    }
  }
}
```

La salida de este programa es idéntica a la del ejemplo anterior.

throw

Hasta el momento, se han capturado excepciones lanzadas por el intérprete Java. Sin embargo, también el propio programa puede lanzar explícitamente una excepción mediante la sentencia **throw**. La forma general de esta sentencia es la siguiente:

throw *objetoThrowable;*

Donde *objetoThrowable* debe ser un objeto del tipo **Throwable** o una subclase de **Throwable**. No se pueden utilizar como excepciones tipos sencillos como **int** o **char**, ni tampoco clases **String** y **Object** que no son **Throwable**. Se puede obtener un objeto de la clase **Throwable** de dos formas: utilizando un parámetro en la cláusula **catch**, o creando un nuevo objeto con el operador **new**.

La ejecución del programa se para inmediatamente después de una sentencia **throw**; y cualquiera de las sentencias que siguen no se ejecutarán. A continuación se inspecciona el bloque **try** más próximo que la encierra, para ver si contiene una sentencia **catch** que coincida con el tipo de excepción. Si es así, el control se transfiere a esa sentencia. Si no, se inspecciona el siguiente bloque **try** que la engloba, y así sucesivamente. Si no se encuentra una sentencia **catch** cuyo tipo coincida con el de la excepción, entonces el gestor de excepciones por omisión interrumpe el programa e imprime el trazado de la pila.

A continuación se presenta un programa de ejemplo que crea y lanza una excepción. El gestor que captura la excepción la relanza al gestor más externo.

```
// Ejemplo de la sentencia throw.
class ThrowDemo (
  static void demoproc() {
    try {
      throw new NullPointerException("demo");
    } catch(NullPointerException e) {
      System.out.println("!Capturada dentro de demoproc.");
      throw e; // se relanza la excepción
    }
  }

  public static void main(String args[]) {
    try {
      demoproc () ;
    } catch(NullPointerException e) {
      System.out.println("Recapturada: " + e);
    }
  }
}
```

Este programa tiene dos oportunidades para tratar el mismo error. En la primera, el método **main( )** establece un contexto de excepción, y a continuación llama a **demoproc( )**. El método **demoproc( )** entonces establece otro contexto de gestión de excepciones e inmediatamente lanza una nueva instancia de **NullPointerException**, que se captura en la siguiente línea. Entonces la excepción se relanza. La salida resultante es la siguiente:

```
Capturada dentro de demoproc.
Recapturada: java.lang.NullPointerException: demo
```

El programa también ilustra cómo se crea uno de los objetos de la clase Exception estándar de Java. Preste especial atención a la siguiente línea:

```
throw new NullPointerException("demo");
```

Aquí, **new** se utiliza para construir una instancia de **NullPointerException**. Todas las excepciones incorporadas por Java en el tiempo de ejecución tienen al menos dos constructores: uno sin parámetros y el otro con un parámetro del tipo cadena. Cuando se utiliza la segunda forma, el argumento especifica una cadena que describe la excepción. Cuando se usa el objeto como argumento de un **print( )** o un **println( )** se imprime esta cadena. También se puede obtener el texto de la cadena mediante una llamada al método **getMessage( )**, que está definido por la clase **Throwable**.

throws

Si un método puede dar lugar a una excepción que no es capaz de gestionar él mismo, se debe especificar este comportamiento de forma que los métodos que llamen al primero puedan protegerse contra esa excepción. Para ello se incluye una cláusula **throws** en la declaración del método. Una cláusula **throws** da un listado de los tipos de excepciones que el método podría lanzar. Esto es necesario para todas las excepciones, excepto las del tipo **Error** o **RuntimeException**, o cualquiera de sus subclases.

Todas las demás excepciones que un método puede lanzar se deben declarar en la cláusula **throws**. Si esto no se hace así, el resultado es un error de compilación.

La forma general de la declaración de un método que incluye una sentencia **throws** es la siguiente:

tipo nombre_ método (lista_de_parámetros) throws lista_de_excepciones
{
// cuerpo del método
}

Donde, *lista_de_excepciones* es una lista de las excepciones que el método puede lanzar, separadas por comas.

A continuación se muestra un programa incorrecto que trata de lanzar una excepción que no captura. Como el programa no especifica una sentencia **throws** que declare este hecho, no se compilará.

```
// Este programa contiene un error y no compilará.
class ThrowsDemo {
  static void throwOne() {
    System.out.println("Dentro de throwOne.");
    throw new IllegalAccessException("demo"):
  }
  public static void main(String args[] ) {
    throwOne():
  }
}
```

Es necesario hacer dos cambios para conseguir que este ejemplo compile. El primero consiste en declarar que **throwOne( )** lanza **IllegalAccessException**. El segundo es que **main( )** debe definir una sentencia **try/catch** que capture esta excepción.

El ejemplo anterior corregido se muestra a continuación:

```
// Ahora es correcto.
class ThrowsDemo {
  static void throwOne() throws IllegalAccessException {
    System.out.println("Dentro de throwOne.");
    throw new IllegalAccessException("demo"):
  }
  public static void main(String args[]) {
    try {
      throwOne ();
    } catch (IllegalAccessException e) {
      System.out.println("Capturada" + e);
    }
  }
}
```

La salida generada por la ejecución de este programa es la siguiente:

```
Dentro de throwOne
Capturada java.lang.IllegalAccessException: demo
```

finally

Cuando se lanzan excepciones, la ejecución dentro de un método sigue un camino no lineal y bastante brusco que altera el flujo normal. Dependiendo de cómo se haya codificado el método, puede incluso suceder que una excepción provoque que el método finalice de forma prematura. Esto puede ser un problema en algunos casos. Por ejemplo, si un método abre un archivo cuando comienza y lo cierra cuando finaliza, entonces no se puede permitir que el mecanismo de gestión de excepciones omita la ejecución del código que cierra el archivo. La palabra clave **finally** está diseñada para resolver este tipo de contingencias.

finally crea un bloque de código que se ejecutará después de que se haya completado un bloque **try/catch** y antes de que se ejecute el código que sigue al bloque **try/catch**. El bloque **finally** se ejecuta independientemente de que se haya lanzado o no alguna excepción. Si se ha lanzado una excepción, el bloque **finally** se ejecuta, incluso aunque ninguna sentencia **catch** coincida con la excepción. Cuando un método está a punto de devolver el control al método llamante desde dentro de un bloque **try/catch** por medio de una excepción no capturada o de una sentencia **return** explícita, se ejecuta también la cláusula **finally** justo antes de que el método devuelva el control. Esta acción tiene utilidad para cerrar descriptores de archivos o liberar cualquier otro recurso que se hubiera asignado al comienzo de un método con la intención de liberarlo antes de devolver el control. La cláusula **finally** es opcional. Sin embargo, cada sentencia **try** requiere, al menos, una sentencia **catch** o **finally**.

El siguiente programa muestra tres métodos distintos, que finalizan en tres diferentes formas, ninguno sin ejecutar sus respectivas sentencias **finally:**

```
// Demostración de finally.
class FinallyDemo {
  // A través de una excepción exterior al método.
  static void procA() {
    try {
      System.out.println("Dentro de procA");
      throw new RuntimeException("demo");
    } finally {
      System.out.println("finally de procA ");
    }
  }

  // Se devuelve el control desde un bloque.
  static void procB() {
    try {
      System.out.println("Dentro de procB");
      return;
      } finally {
        System.out.println("finally de procB");
      }
    }
// Ejecución normal de un bloque try.
  static void procC() {
    try {
      System.out.println("Dentro de procC");
      } finally {
```

```
      System.out.println("finally de procC");
    }
  }

  public static void main(String args[]) {
    try {
      procA () ;
    } catch (Exception e) {
      System.out.println("Excepción capturada");
    }
    procB () ;
    procC () ;
  }
}
```

En este ejemplo, **procA( )** sale prematuramente del bloque **try** lanzando una excepción. La sentencia **finally** se ejecuta durante la salida. En el método **procB( )** se sale del bloque **try** por medio de la sentencia **return**. La sentencia **finally** se ejecuta antes de que el método **procB( )** devuelva el control. En el método **procC( )**, la sentencia **try** se ejecuta normalmente, sin error. Sin embargo, sí se ejecuta el bloque **finally.**

NOTA *Si se asocia un bloque finally con un bloque try, el bloque finally se ejecuta cuando concluye el bloque try.*

La salida generada por el programa anterior es la siguiente:

```
Dentro de procA
finally de procA
Excepción capturada
Dentro de procB
finally de procB
Dentro de procC
finally de procC
```

Excepciones integradas en Java

Dentro del paquete estándar **java.lang**, Java define varias clases de excepciones. Algunas ya se han usado en los ejemplos anteriores. Las excepciones más comunes son subclases del tipo estándar **RuntimeException**. Como se explicó antes estas excepciones no necesitan ser incluidas en la lista **throws** de ningún método. Dentro del lenguaje Java, se denomina *excepciones no comprobadas* a estas excepciones, ya que el compilador no controla si el método gestiona o lanza estas excepciones. Las excepciones de este tipo definidas en **java.lang** se detallan en la Tabla 10.1. La Tabla 10.2 muestra un listado de las excepciones definidas por **java.lang** que deben ser incluidas en la lista **throws** de un método si ese método puede generar una de estas excepciones y no puede gestionarla por sí mismo. A estas excepciones se denomina *excepciones comprobadas.* Java define muchos otros tipos de excepciones en diversos paquetes de su biblioteca de clases.

Excepciones	Significado
ArithmeticException	Error aritmético, como, por ejemplo, división entre cero.
ArrayIndexOutOfBoundsException	Índice del arreglo fuera de su límite o rango.
ArrayStoreException	Se ha asignado a un elemento de un arreglo un tipo incompatible.
ClassCastException	Conversión de tipo inválido.
IllegalArgumentException	Uso inválido de un argumento al llamar a un método.
IllegalMonitorStateException	Operación de monitor inválida, tal como esperar un hilo no bloqueado.
IllegalStateException	El entorno o aplicación están en un estado incorrecto o inválido.
IllegalThreadStateException	La operación solicitada es incompatible con el estado actual del hilo.
IndexOutOfBoundsException	Algún tipo de índice está fuera de rango o de su límite.
NegativeArraySizeException	Arreglo creado con un tamaño negativo.
NullPointerException	Uso incorrecto de una referencia a null.
NumberFormatException	Conversión incorrecta de un valor tipo string a un formato numérico.
SecurityException	Intento de violación de seguridad.
StringIndexOutOfBounds	Intento de sobrepasar el límite o rango de un valor string.
UnsupportedOperationException	Operación no soportada.

TABLA 10.1 Excepciones no comprobadas definidas en **java.lang** como subclases de **RuntimeException**

Excepciones	Significado
ClassNotFoundException	No se ha encontrado la clase.
CloneNotSupportedException	Intento de duplicación de un objeto que no implementa la interfaz **Cloneable**.
IllegalAccessException	Se ha denegado el acceso a una clase.
InstantiationException	Intento de crear un objeto de una clase abstracta o interfaz.
InterruptedException	Hilo interrumpido por otro hilo.
NoSuchFieldException	No existe el campo solicitado.
NoSuchMethodException	No existe el método solicitado.

TABLA 10.2 Excepciones comprobadas definidas en **java.lang**

Creando excepciones propias

Aunque las excepciones del núcleo de Java gestionan la mayor parte de los errores habituales, nosotros podríamos desear crear nuestros propios tipos de excepciones para tratar situaciones específicas que se presenten en nuestras aplicaciones. Esto se puede hacer de forma bastante

sencilla, definiendo una subclase de la clase **Exception**, que es por supuesto, una subclase de **Throwable**. No es necesario que estas subclases creadas por el usuario implementen nada; simplemente, su existencia en el sistema nos permitirá usarlas como excepciones.

La clase **Exception** no define por sí misma método alguno, pero hereda, evidentemente, los métodos que proporciona la clase **Throwable**. Además, todas las excepciones, incluyendo las creadas por nosotros, pueden disponer de los métodos definidos por la clase **Throwable**. Dichos métodos se muestran en la Tabla 10.3.

Método	Descripción
Throwable fillInStackTrace( )	Devuelve un objeto de la clase **Throwable** que contiene el trazado completo de la pila. Este objeto puede volver a ser lanzado.
Throwable getCause( )	Devuelve la excepción subyacente a la excepción actual. Si no existe una excepción subyacente devuelve **null**.
String getLocalizedMessage( )	Devuelve una cadena con la descripción localizada de la excepción.
String getMessage( )	Devuelve la descripción de la excepción.
StackTraceElement[ ] getStackTrace( )	Devuelve un arreglo que contiene el trazado de la pila, un elemento a la vez, el arreglo es de tipo **StackTraceElement**. El método en la parte superior de la pila es el último método llamado antes de que la excepción fuera lanzada. Este método se localiza en la primera posición del arreglo. La clase **StackTraceElement** da al programa acceso a la información de cada elemento, como por ejemplo el nombre del método.
Throwable initCause (Throwable causeExc)	Asocia la referencia *causeExc* con la excepción invocada como la causa de la misma. Regresa una referencia a la excepción.
void printStackTrace( )	Presenta en pantalla el trazado de la pila
void printStackTrace(PrintStream stream)	Envía el trazado de la pila a un determinado flujo.
printStackTrace(PrintWriter stream)	Envía el trazado de la pila a un determinado flujo.
void setStackTrace(StackTraceElement elements[ ])	Coloca en la pila los elementos especificados en el arreglo elements. Este método es para aplicaciones especializadas, no para uso convencional.
String toString( )	Devuelve una cadena con la descripción de la excepción. Este método es llamado por println( ) cuando se desea imprimir un objeto de la clase **Throwable**.

TABLA 10.3 Métodos definidos por la clase **Throwable**

Además, estos métodos se pueden sobrescribir en las clases de excepción propias.

La clase **Exception** define cuatro constructores. Dos de ellos incluidos por JDK 1.4 para soportar excepciones encadenadas, se describen en la siguiente sección. Los otros dos se describen aquí:

Exception ( )

Exception (String *msg*)

La primera forma crea una excepción sin descripción. La segunda forma nos permite especificar una descripción para la excepción.

Aunque especificar una descripción cuando se crea una excepción es útil, alguna veces es mejor sobrescribir al método **toString( )**, esto debido a que la versión de **toString( )** definida por la clase **Throwable** y heredada por la clase **Exception** primero despliega el nombre de la excepción seguido de dos puntos y en seguida la descripción de la excepción dada en el constructor. Al sobrescribir el método **toString( )** es posible evitar que se muestre el nombre de la excepción y los dos puntos, con ello podemos generar una salida más limpia, deseable en algunos casos.

En el siguiente ejemplo se declara una subclase de la clase **Exception** que se usa posteriormente para señalar una condición de error en un método. Dicha subclase sobrescribe el método **toString( )** para poder imprimir una descripción cuidadosamente adaptada de la excepción.

```
// Este programa crea un tipo de excepción propio.
class MiExcepcion extends Exception {
  private int detalle;

  MiExcepcion (int a) {
    detalle = a;
  }

  public String toString() {
    return " MiExcepcion [" + detalle + "]";
  }
}

class ExcepcionDemo (
  static void compute(int a) throws MiExcepcion {
    System.out.println("Ejecuta compute(" + a + ")");
      if(a > 10)
        throw new MiExcepcion(a);
      System.out.println("Finalización normal");

  public static void main(String args[]) {
    try {
      compute (1);
      compute (20) ;
    } catch (MiExcepcion e) {
      System.out.println("Captura de: " + e);
    }
  }
}
```

En este ejemplo se define una subclase de **Exception** llamada **MiExcepcion**. Esta subclase es muy sencilla: tiene únicamente un constructor y un método sobrecargado, **toString( )**, que permitirá presentar el valor de la excepción.

La clase **ExcepcionDemo** define un método llamado **compute( )** y que lanza un objeto del tipo **MyException**. La excepción se lanza cuando el parámetro entero del método **compute( )** es mayor que 10. El método **main( )** establece un gestor de excepciones para **MiExcepcion**, y a continuación llama al método **compute( )** con un valor válido del parámetro, es decir, menor que 10, y con un valor no válido, para mostrar los dos caminos que sigue el código. El resultado es el siguiente:

```
Ejecuta compute(1)
Finalización normal
Ejecuta compute(20)
Captura MiExcepcion[20]
```

Excepciones encadenadas

A partir del JDK 1.4, una nueva característica se ha incorporado en el subsistema de excepciones: las *excepciones encadenadas*. La característica de excepción encadenada nos permite asociar una excepción con otra. Esta segunda excepción describe la causa de la primera excepción. Por ejemplo, imaginemos una situación en la cual un método lanza una excepción del tipo **ArithmeticException** debido a un intento de división entre cero. Sin embargo, la causa real del problema fue la ocurrencia de un error de E/S la cual causa que el divisor reciba un valor inapropiado. Aunque el método lanzará una excepción **ArithmeticException**, debido a que ese es el error que ha ocurrido, podríamos además desear que el código que llamó al método conozca que la causa subyacente fue un error de E/S. Las excepciones encadenadas nos permiten gestionar ésta y otras situaciones en las cuales existen capas o niveles de excepciones.

Para crear excepciones encadenadas se añadieron dos métodos y dos constructores a la clase **Throwable**. Los constructores son:

Throwable (Throwable *exc*)
Throwable (String *msg,* Throwable *exc*)

En la primera forma, *exc* es la excepción que causa a la excepción actual. Esto es, *exc* es la razón subyacente a la ocurrencia de la nueva excepción. La segunda forma nos permite especificar una descripción al mismo tiempo que se especifica la excepción causante de la actual. Estos dos constructores también han sido añadidos a las clases **Error**, **Exception** y **RuntimeException**.

Los métodos de encadenado de excepciones añadidos a la clase **Throwable** son **getCause( )** e **initCause( )**. Estos métodos se muestran en la Tabla 10-3 y se repiten aquí

Throwable getCause( )
Throwable initCause(Throwable *exc*)

El método **getCause( )** regresa la excepción subyacente a la excepción actual. Si no existe una excepción subyacente regresa **null**. El método **initCause( )** asocia *exc* con la excepción que realiza la invocación y regresa una referencia a la excepción. Así podemos asociar una causa con una excepción después de que la excepción ha sido creada. Sin embargo, la excepción causante puede ser asociada sólo una vez. Por ello, es posible llamar a **initCause( )** sólo una vez para cada objeto de excepción. Si la excepción causante fue establecida por un constructor, no es posible establecerla de nuevo utilizando **initCause( )**. En general, **initCause( )** es utilizada para asignar una causa a excepciones cuyas clases tipo no cuentan con los dos constructores adicionales descritos antes.

Veamos un ejemplo que ilustra el mecanismo de gestión de excepciones encadenadas:

```
// Ejemplo de excepciones encadenadas
class ExcepcionEncadenadaDemo {
  static void demoproc() {

    // crea una excepción
    NullPointerException e =
      new NullPointerException("capa superior");
```

```
      // añadir una causa
      e.initCause(new ArithmeticException("causa"));

      throw e;
    }

  public static void main(String args[]) {
    try {
      demoproc() ;
    } catch (NullPointerException e) {
      // mostrar la excepción superior
      System.out.println("Atrapada: " + e);

      // mostrar la excepción causante
      System.out.println ("Causa Original: " +
                          e.getCause() );
    }
  }
}
```

La salida resultante de la ejecución del programa anterior es:

```
Atrapada: java.lang.NullPointerException: capa superior
Causa Original: java.lang.ArithmeticException: causa
```

En este ejemplo, la excepción de nivel superior es **NullPointerException**. A ésta se añade una excepción de tipo **ArithmeticException** como causante. Cuando la excepción es lanzada fuera del método **demoproc( )**, es atrapada por el método **main( )**. Ahí, la excepción de nivel superior es mostrada seguida por la excepción subyacente, la cual es obtenida mediante la llamada al método **getCause( ).**

Las excepciones encadenadas pueden ser continuadas con cualquier profundidad necesaria. Así, la excepción causa puede a su vez tener una excepción causante. Aunque una cadena excesivamente larga de excepciones puede ser signo de un pobre diseño del sistema.

Las excepciones encadenadas no son algo que todo programa necesite. Sin embargo, en casos en los cuales el conocimiento de una causa subyacente es útil las excepciones encadenadas ofrecen una solución elegante.

Utilizando excepciones

La gestión de excepciones proporciona un mecanismo muy potente para controlar programas complejos, con muchas características dinámicas, durante la ejecución. Es importante considerar a **try**, **throw** y **catch** como formas limpias de gestionar errores y problemas inesperados en la lógica de un programa. A diferencia de otros lenguajes en los cuales se acostumbra devolver un código de error cuando se produce un fallo, Java utiliza excepciones. Así cuando un método puede fallar debemos hacer que lance una excepción. Ésta es una manera más limpia de tratar los modos de fallo.

Una última cuestión que se ha de tener en cuenta sobre la gestión de excepciones en Java, es que no se debe considerar este mecanismo como otra vía para realizar ramificaciones, ya que si se hace así, lo que se consigue es crear un código que puede resultar finalmente incomprensible y de difícil mantener.

11 CAPÍTULO

Programación multihilo

A diferencia de la mayoría de los lenguajes de programación, Java proporciona soporte para la *programación multihilo*. Un programa multihilo contiene dos o más partes que pueden ser ejecutadas de manera concurrente o simultánea. Cada parte del programa se denomina *hilo*, y cada hilo define un camino de ejecución independiente. Por lo tanto, la programación multihilo es una forma especializada de multitarea.

Probablemente esté familiarizado con la multitarea, ya que casi todos los sistemas operativos modernos la permiten. Sin embargo, existen dos tipos distintos de multitarea: la basada en procesos y la basada en hilos. Es importante comprender la diferencia entre las dos. Para la mayoría de lectores la forma más familiar es la multitarea basada en el proceso. Un *proceso* es, en esencia, un programa que se está ejecutando. Por lo tanto, la multitarea *basada en procesos* es la característica que permite a su computadora ejecutar dos o más programas concurrentemente. Por ejemplo, la multitarea basada en procesos permite que se ejecute el compilador Java al mismo tiempo que se está utilizando el editor de textos. En la multitarea basada en procesos, un programa es la unidad más pequeña de código que el sistema de cómputo puede gestionar.

En un entorno de multitarea *basada en hilos*, el hilo es la unidad de código más pequeña que se puede gestionar. Esto significa que un sólo programa puede realizar dos o más tareas simultáneamente. Por ejemplo, un editor de textos puede dar formato a un texto al mismo tiempo que está imprimiendo, ya que estas dos acciones las realizan dos hilos distintos. Por tanto, la multitarea basada en procesos actúa sobre "tareas generales", mientras que la multitarea basada en hilos gestiona los detalles.

La multitarea basada en hilos requiere un menor costo de operación que la basada en procesos. Los procesos son tareas más pesadas, es decir requieren más recursos, y necesitan espacio de direccionamiento propio. La comunicación entre procesos es costosa y limitada. El intercambio de contextos al pasar de un proceso a otro también es costoso. Los hilos, por otra parte, son tareas ligeras. Comparten el mismo espacio de direccionamiento y el mismo proceso. Tanto la comunicación entre hilos como el intercambio de contextos de un hilo al próximo tienen un costo bajo. Los programas de Java utilizan entornos de multitarea basada en procesos, pero la multitarea basada en procesos no está bajo el control de Java. Sin embargo, la multitarea basada en hilos sí.

La multitarea basada en hilos permite escribir programas muy eficientes que hacen uso óptimo del CPU, ya que el tiempo que éste está libre se reduce al mínimo. Esto es especialmente importante en los entornos interactivos de red en los que Java funciona, donde el tiempo libre del CPU es común. Por ejemplo, la velocidad de transmisión de datos en la red es mucho más baja que la velocidad de proceso de la computadora. Incluso la velocidad de lectura y escritura en el sistema

local de archivos es mucho más baja que la velocidad de proceso del CPU. Naturalmente, la velocidad con que el usuario introduce la información, es mucho más baja que la velocidad de la computadora. En un entorno tradicional de un solo hilo, el programa tiene que esperar a que se realicen cada una de estas tareas antes de procesar la siguiente, aunque el CPU esté inactivo la mayor parte del tiempo. La multitarea basada en hilos permite acceder y aprovechar este tiempo de inactividad del CPU.

Si ya ha programado en sistemas operativos tales como Windows, entonces ya conoce la programación multihilo. Sin embargo, la forma en que Java maneja los hilos hace que la programación multihilo sea especialmente simple, ya que muchos de los detalles son gestionados por Java y no por el programador.

El modelo de hilos en Java

El intérprete de Java depende de los hilos en muchos aspectos, y todas las bibliotecas de clases se han diseñado teniendo en cuenta el modelo multihilo. De hecho, Java utiliza los hilos para permitir que todo el entorno sea asíncrono. Esto aumenta la eficacia, impidiendo el desaprovechamiento de ciclos del CPU.

El valor del entorno multihilo se comprende mejor cuando se compara con el otro modo de funcionamiento. Los sistemas de un solo hilo utilizan un enfoque denominado *ciclo de evento* con *sondeo*. En este modelo, un solo hilo de control se ejecuta en un ciclo infinito, sondeando una única cola de eventos para decidir cuál se procesará a continuación. Una vez que este mecanismo de sondeo obtiene una señal que indica que un archivo de la red está listo para ser leído, entonces el ciclo de evento selecciona el gestor de control apropiado para ese evento. Hasta que este gestor regrese el control, nada más puede ocurrir en el sistema, y esto supone un desaprovechamiento del CPU. Puede ocurrir también que una parte de un programa domine el sistema e impida que otros eventos sean procesados. En general, en un entorno de un solo hilo, cuando un hilo *bloquea* (es decir, suspende la ejecución) porque está esperando algún recurso, el programa entero se detiene.

La ventaja de la programación multihilo en Java es que se elimina el mecanismo principal de ciclo/sondeo. Un hilo puede detenerse sin paralizar el resto de las partes del programa. Por ejemplo, el tiempo de inactividad que se produce cuando un hilo lee datos de la red o espera a que el usuario introduzca una información, puede ser aprovechado por otro. La programación multihilo permite que los ciclos de animación paren durante un segundo entre una imagen y la siguiente sin que todo el sistema se detenga. Cuando en un programa Java, un hilo se bloquea, solo ese hilo se detiene y todos los demás continúan su ejecución.

Los hilos pueden encontrarse en distintos estados. Un hilo puede estar *ejecutándose* o *preparado para ejecutarse* tan pronto como disponga de tiempo de CPU. Un hilo que está ejecutándose puede estar suspendido, lo que significa que temporalmente se suspende su actividad. Un hilo suspendido puede *reanudarse*, permitiendo que continúe su tarea donde la dejó. Un hilo puede estar bloqueado cuando está esperando un determinado recurso. En cualquier instante, un hilo puede *detenerse*, finalizando su ejecución de forma inmediata. Una vez detenido, un hilo no puede reanudarse.

Prioridades en hilos

Java asigna a cada hilo una prioridad que determina cómo se debe tratar ese hilo en comparación con los demás. Las prioridades de los hilos son valores enteros que especifican la

prioridad relativa de un hilo sobre otro. Como valor absoluto, una prioridad no tiene sentido alguno; un hilo de prioridad más alta no se ejecuta más rápidamente que otro de prioridad más baja si es el único hilo que se está ejecutando. La prioridad de un hilo se utiliza para decidir cuándo se debe pasar de la ejecución de un hilo a la del siguiente. A esto se denomina *cambio de contexto*. Las reglas que determinan cuándo debe tener lugar un cambio de contexto son muy sencillas:

- *Un hilo puede ceder voluntariamente el control.* Esto se hace por abandono explícito, al quedarse dormido, o bloqueado por una E/S pendiente. Cuando esto ocurre, se examinan todos los demás hilos, y se le asigna el CPU al que esté preparado para ejecutarse y tenga la más alta prioridad.
- *Un hilo puede ser desalojado por otro con prioridad más alta.* En este caso, un hilo de prioridad más baja que no libera el procesador es simplemente desalojado, sin tener en cuenta lo que esté haciendo, por otro de prioridad más alta. Básicamente, cuando un hilo de prioridad más alta desee ejecutarse, lo hará. A esto se denomina *multitarea por desalojo*.

Una situación un poco más complicada es la que se produce cuando dos hilos de la misma prioridad compiten por el CPU. En sistemas operativos como Windows, los hilos con la misma prioridad se reparten automáticamente el tiempo mediante un algoritmo circular (round-robin). Para otros sistemas operativos, los hilos deben ceder voluntariamente el control a otros de la misma prioridad. Si no lo hacen así, estos últimos no se ejecutarán.

PRECAUCIÓN *Las diferentes formas en que los distintos sistemas operativos realizan la conmutación de contexto de hilos con la misma prioridad pueden ocasionar problemas de portabilidad.*

Sincronización

La programación multihilo introduce un comportamiento asíncrono en los programas, aunque en ocasiones puede ser necesario el sincronismo. Esto sucede, por ejemplo, si se quiere que dos hilos se comuniquen y compartan una estructura complicada de datos, como una lista enlazada. En este caso será necesario asegurar que los dos hilos no entren en conflicto. Esto es, impedir que uno de los hilos escriba mientras el otro está realizando la lectura. Java implementa para este propósito un elegante modelo clásico de sincronización entre procesos, el monitor. El monitor es un mecanismo de control que fue definido por primera vez por C. A. R. Hoare. Se puede considerar un monitor como una pequeña caja que contiene solamente un hilo. Una vez que un hilo entra en el monitor, todos los demás hilos deben esperar hasta que el hilo salga del monitor. De esta forma, un monitor se puede utilizar para proteger el hecho de que varios hilos manipulen al mismo tiempo un recurso.

La mayor parte de los sistemas multihilo presentan los monitores como objetos que los programas deben obtener y manipular explícitamente. Java proporciona una solución más clara. No existe la clase "Monitor"; en su lugar, cada objeto tiene su propio monitor implícito que se introduce automáticamente cuando se llama a uno de los métodos sincronizados del objeto. Cuando un hilo está dentro de un método sincronizado, ningún otro hilo puede llamar a ningún otro método sincronizado del mismo objeto. Esto permitirá escribir un código multihilo muy claro y conciso, debido a que el soporte para la sincronización se encuentra establecido dentro del lenguaje.

Intercambio de mensajes

Después de dividir el programa en distintos hilos, es necesario definir cómo se comunicarán entre sí. Cuando se programa en otros lenguajes, existe una dependencia del sistema operativo para establecer la comunicación entre hilos, y esto, evidentemente, añade mayor costo de ejecución. En contraste, Java proporciona una forma limpia y de bajo costo que permite la comunicación entre dos o más hilos, por medio de llamadas a métodos predefinidos que tienen todos los objetos. El sistema de mensajes de Java permite que un hilo entre en un método sincronizado de un objeto, y espere ahí hasta que otro hilo le notifique explícitamente que debe salir.

La clase Thread y la interfaz Runnable

El sistema multihilo de Java está construido en torno a la clase **Thread**, sus métodos, y su correspondiente interfaz, **Runnable**. La clase **Thread** encapsula a un hilo de ejecución. Debido a que no se puede hacer referencia directamente al estado de un hilo en ejecución, es necesario utilizar una instancia de la clase **Thread** que represente a dicho hilo. Para crear un nuevo hilo, el programa deberá extender la clase **Thread** o bien implementar la interfaz **Runnable**.

La clase **Thread** define varios métodos que ayudan a la gestión de los hilos. A continuación, se muestran los métodos que se utilizarán en este capítulo:

Método	Significado
getName	Obtiene el nombre de un hilo.
getPriority	Obtiene la prioridad de un hilo.
isAlive	Determina si un hilo todavía se está ejecutando.
join	Espera la terminación de un hilo.
run	Punto de entrada de un hilo.
sleep	Suspende un hilo durante un periodo de tiempo.
start	Comienza un hilo llamando a su método run.

Hasta el momento, todos los ejemplos de este libro han utilizado un solo hilo de ejecución. El resto del capítulo explica cómo funcionan la clase **Thread** y la interfaz **Runnable** para crear y gestionar hilos, comenzando con el hilo que tienen todos los programas de Java: el hilo principal.

El hilo principal

Cuando un programa Java comienza su ejecución, hay un hilo ejecutándose inmediatamente. Este hilo se denomina normalmente *hilo principal del programa*, porque es el único que se ejecuta al comenzar el programa. El hilo principal es importante por dos razones:

- Es el hilo a partir del cual se crean el resto de los hilos del programa.
- Normalmente, debe ser el último que finaliza su ejecución debido a que es el responsable de realizar diversas acciones de cierre.

Aunque el hilo principal se crea automáticamente cuando el programa comienza, se puede controlar a través de un objeto **Thread**. Para ello, se debe obtener una referencia al mismo

llamando al método **currentThread( )**, que es un miembro **public static** de la clase **Thread**. Su forma general es la siguiente:

```
static Thread currentThread( )
```

Este método devuelve una referencia al hilo desde donde fue llamado. Una vez obtenida la referencia del hilo principal, se le puede controlar del mismo modo que a cualquier otro hilo.

Comencemos revisando el siguiente ejemplo:

```
// Control del hilo principal.
class DemoHiloActual {
  public static void main (String args[]) {
    Thread t = Thread.currentThread();

    System.out.println ("Hilo actual: " + t);

    // Cambio del nombre del hilo
    t.setName ("Mi Hilo");
    System.out.println ("Después del cambio de nombre: " + t);

    try {
      for (int n = 5; n > 0; n--) {
        System.out.println (n);
        Thread.sleep(1000);
      }
    } catch (InterruptedException e) {
      System.out.println ("Interrupción del hilo principal");
    }
  }
}
```

En este programa, se obtiene una referencia al hilo actual (el hilo principal en este caso) llamando al método **currentThread( )**, dicha referencia es almacenada en la variable local **t**. A continuación, el programa visualiza la información referente al hilo. Entonces, el programa llama al método **setName( )** para cambiar el nombre interno del hilo, y se visualiza de nuevo la información referente al hilo.

Luego, un ciclo cuenta de forma descendente desde cinco, con una pausa de un segundo entre cada línea. Esta pausa se obtiene utilizando el método **sleep( )**. El argumento del método **sleep( )** especifica el retraso en milisegundos. Observe el bloque **try /catch** en el que se encuentra el ciclo. El método **sleep( )** de **Thread** podría lanzar una excepción **InterruptedException**. Esto es lo que ocurriría si algún otro hilo intentase interrumpir a éste mientras está dormido. En el ejemplo, si se le interrumpe se imprime un mensaje. En un programa real, sería necesario resolver la situación de una forma distinta. La salida generada por el programa es:

```
Hilo actual: Thread[main,5,main]
Después del cambio de nombre: Thread[Mi Hilo,5,main]
5
4
3
2
1
```

Observe la salida que se produce cuando se usa **t** como argumento de **println( )**. Aparecen en orden: el nombre del hilo, su prioridad y el nombre de su grupo. Por omisión, el nombre del hilo principal es **main**. Su prioridad es 5, que es el valor por omisión, y **main** es también el nombre del grupo de hilos a los que pertenece este hilo. Un *grupo de hilos* es una estructura de datos que controla el estado de una colección de hilos como un todo. Después de cambiar el nombre del hilo, se vuelve a presentar **t**. Esta vez se imprime el nuevo nombre del hilo.

Analicemos con más detenimiento los métodos definidos por **Thread** que son utilizados en el programa. El método **sleep( )** hace que se suspenda la ejecución del hilo desde el que fue llamado durante un periodo de tiempo especificado en milisegundos. Su forma general es la siguiente:

```
static void sleep (long milisegundos) throws InterruptedException
```

El número de milisegundos que se suspende la ejecución se especifica en la variable milisegundos. Este método puede lanzar una excepción **InterruptedException**.

El método **sleep( )** tiene una segunda forma, que se muestra a continuación, y que permite especificar el periodo de tiempo en términos de milisegundos y nanosegundos:

```
static void sleep (long milisegundos, int nanosegundos) throws InterruptedException
```

Esta segunda forma es útil en entornos que permitan periodos de tiempo tan cortos como el nanosegundo.

Como se ha visto en el programa anterior, se puede asignar un nombre a un hilo con el método **setName( )**. También se puede obtener el nombre de un hilo llamando al método **getName( )** (este procedimiento no se muestra en el programa). Estos métodos son miembros de la clase **Thread** y se declaran de la siguiente forma:

```
final void setName (String nombreHilo)

final String getName( )
```

Donde *nombreHilo* especifica el nombre del hilo.

Creación de un hilo

En un sentido amplio, se puede crear un hilo creando un objeto del tipo **Thread**. Java define dos formas en la que se puede hacer esto:

- Implementando la interfaz **Runnable**.
- Extendiendo la propia clase **Thread**.

Los dos siguientes apartados analizan cada una de estas opciones.

Implementación de la interfaz Runnable

La forma más fácil de crear un hilo es crear una clase que implemente la interfaz **Runnable**. Esta interfaz permite abstraer el concepto de una unidad de código ejecutable. Se puede construir un hilo sobre cualquier objeto que implemente la interfaz **Runnable**. Para ello, una clase necesita implementar un único método llamado **run( )**, que se declara de la siguiente forma:

public void run( )

Dentro del método **run( )**, se definirá el código que constituye el nuevo hilo. Es importante entender que **run( )** puede llamar a otros métodos, usar otras clases y declarar variables de la misma forma que el hilo principal. La única diferencia es que el método **run( )** establece el punto de entrada para otro hilo de ejecución concurrente dentro del programa. Este hilo finalizará cuando el método **run( )** devuelva el control.

Después de crear una clase que implemente la interfaz **Runnable**, se creará un objeto del tipo **Thread** dentro de esa clase. **Thread** define varios constructores. A continuación se presenta el que se usa en este ejemplo:

Thread (Runnable *objetoHilo*, String *nombreHilo*)

En este constructor, *objetoHilo* es una instancia de una clase que implementa la interfaz **Runnable** y define el punto en el que comenzará la ejecución del hilo. La variable *nombreHilo* indica el nombre del nuevo hilo.

El nuevo hilo que se acaba de crear no comenzará su ejecución hasta que se llame al método **start( )**, declarado dentro de **Thread**. Esencialmente, **start( )** ejecuta una llamada a **run( )**. A continuación se muestra el método **start( )**:

void start( )

En el ejemplo siguiente se crea un nuevo hilo y se inicia su ejecución:

```
// Creación de un segundo hilo.
class NewThread implements Runnable {
  Thread t;

  NewThread () {
    //Crea el segundo hilo
    t = new Thread (this, "Hilo demo");
    System.out.println ("Hilo hijo: " + t);
    t.start(); // Comienzo del hilo
  }

  // Este es el punto de entrada para el segundo hilo.
  public void run() {
    try {
      for (int i = 5; i > 0; i--) {
        System.out.println ("Hilo hijo: " + i);
        Thread.sleep(500);
      }
    } catch (InterruptedException e) {
      System.out.println ("Interrupción del hilo hijo. ");
    }
    System.out.println ("Salida del hilo hijo.");
  }
}

class DemoHilo {
  public static void main (String args[]) {
    new NewThread ();   // creación de un nuevo hilo
```

```
    try {
      for (int i = 5; i > 0; i--) {
        System.out.println ("Hilo principal: " + i);
        Thread.sleep(1000);
      }
    } catch (InterruptedException e) {
      System.out.println ("Interrupción del hilo principal.");
    }
    System.out.println ("Salida del hilo principal.");
  }
}
```

La siguiente sentencia dentro del constructor de **NewThread** sirve para crear objeto **Thread**.

```
t = new Thread (this, "Hilo demo");
```

Pasando **this** como primer argumento, se indica que el nuevo hilo llame al método **run( )**en este objeto. A continuación, se llama al método **start( )** para que comience la ejecución del hilo con el método **run( )**. Esto hace que comience el ciclo **for** del hilo hijo. Después de llamar a **start( )**, el constructor de **NewThread** finaliza volviendo a **main( )**. El hilo principal se reanuda, entrando en el ciclo **for**. A partir de ahí, ambos hilos continúan su ejecución, compartiendo el CPU, hasta que sus respectivos ciclos terminan. La salida que se obtiene al ejecutar este programa es la siguiente (la salida puede variar dependiendo de la velocidad del procesador y la carga de tareas):

```
Hilo hijo: Thread [Hilo demo, 5, main]
Hilo principal: 5
Hilo hijo: 5
Hilo hijo: 4
Hilo principal: 4
Hilo hijo: 3
Hilo hijo: 2
Hilo principal: 3
Hilo hijo: 1
Salida del hilo hijo.
Hilo principal: 2
Hilo principal: 1
Salida del hilo principal.
```

Como ya se ha dicho antes, en un programa multihilo, muchas veces el hilo principal debe ser el último que finalice su ejecución. De hecho, con algunos intérpretes de Java antiguos, si el hilo principal finaliza antes de que algún hilo hijo haya completado su ejecución, entonces el intérprete Java puede "bloquearse". El programa anterior asegura que el hilo principal es el último que finaliza su ejecución, ya que el hilo principal se suspende durante 1000 milisegundos entre cada iteración, mientras que el hilo hijo lo hace solamente durante 500 milisegundos, esto hace que el hilo hijo finalice antes que el hilo principal. En breve veremos una mejor forma de esperar a que un hilo termine.

Extensión de la clase Thread

La segunda forma de crear un hilo es crear una nueva clase que extienda la clase **Thread**, y crear entonces una instancia de esa clase. La nueva clase debe sobrescribir el método **run( )**, que es el

punto de entrada para el nuevo hilo. También debe llamar al método **start( )** para comenzar la ejecución del nuevo hilo. A continuación se presenta el programa anterior, realizando esta vez una extensión de la clase **Thread**:

```
// Creación de un segundo hilo extendiendo la clase Thread
class NewThread extends Thread {

  NewThread() {
    // Creación de un nuevo hilo
    super ("Hilo demo");
    Systern.out.println ("Hilo hijo: " + this);
    start();    // Comienza el hilo
  }

  // Este es el punto de entrada para el segundo hilo.
  public void run () {
    try {
      for (int i = 5; i > 0; i--) {
        Systern.out.println ("Hilo hijo: " + i);
        Thread.sleep(500);
      }
    } catch (InterruptedException e) {
      Systern.out.println ("Interrupción del hilo hijo.");
    }
    System.out.println ("Salida del hilo hijo.");
  }
}

class ExtendThread {
  public static void main (String args[]) {
    new NewThread(); // Creación de un nuevo hilo

    try {
      for (int i = 5; i > 0; i--) {
        Systern.out.println ("Hilo principal: " + i);
        Thread.sleep(l000);
      }
    } catch (InterruptedException e) {
      System.out.println ("Interrupción del hilo principal.");
    }
    System.out.println ("Salida del hilo principal");
  }
}
```

Este programa genera la misma salida que la versión anterior. Como se puede ver, el hilo hijo se crea a través de una instancia de la clase **NewThread**, que es una clase derivada de **Thread**.

Observe que mediante la llamada a **super( )** dentro de **NewThread**, se invoca la siguiente forma del constructor de **Thread**:

public Thread (String *nombreHilo*)

donde la variable *nombreHilo* especifica el nombre del hilo.

Elección de una de las dos opciones

En este momento nos podemos preguntar por qué Java permite crear un hilo hijo de dos formas distintas y cuál de las dos es mejor. Las respuestas a estas preguntas nos llevan al mismo punto. La clase **Thread** define varios métodos que pueden ser, sobrescritos por una clase derivada. De estos métodos, el único que debe ser sobrescrito es el método **run( )**, que es, naturalmente el mismo método que se requiere implementar la interfaz **Runnable**. Muchos programadores de Java opinan que las clases sólo se deben extender cuando van a ser mejoradas o modificadas de alguna forma. Así, si no se va a sobrescribir ningún otro método de la clase **Thread**, probablemente sea mejor implementar la interfaz **Runnable**. Evidentemente esto depende del programador. Sin embargo, en el resto de este capítulo, se crearán hilos utilizando clases que implementen **Runnable**.

Creación de múltiples hilos

Hasta ahora sólo se han usado dos hilos: el hilo principal y un hilo hijo. Sin embargo nuestros programas pueden generar tantos hilos como se requiera. El siguiente programa crea tres hilos hijo:

```
// Creación de múltiples hilos.
class NewThread implements Runnable {
  String name; // nombre del hilo
  Thread t;

  NewThread (String threadname) {
    name = threadname;
    t = new Thread(this, name);
    System.out.println ("Nuevo hilo: " + t);
    t.start(); // Comienza el hilo
  }

  // Este es el punto de entrada del hilo.
  public void run() {
    try {
        for (int i = 5; i > 0; i-) {
        System.out.println (name + ":" + i);
        Thread.sleep(l000);
      }
    } catch (InterruptedException e) {
      System.out.println ("Interrupción del hilo" + name);
    }
    System.out.println (" Salida del hilo" + name);
  }
}

class MultiThreadDemo {
  public static void main (String args[]) {
    new NewThread ("Uno"); // comienzo de los hilos
    new NewThread ("Dos");
    new NewThread ("Tres") ;
```

```
    try {
      // Espera a que los otros hilos terminen
      Thread.sleep(10000);
    } catch (InterruptedException e) {
      System.out.println("Interrupción del hilo principal");
    }
    System.out.println ("Salida del hilo principal");
  }
}
```

La salida de este programa es:

```
Nuevo hilo: Thread [Uno, 5, main]
Nuevo hilo: Thread [Dos, 5, main]
Nuevo hilo: Thread [Tres, 5, main]
Uno: 5
Dos: 5
Tres: 5
Uno: 4
Dos: 4
Tres: 4
Uno: 3
Tres: 3
Dos: 3
Uno: 2
Tres: 2
Dos: 2
Uno: 1
Tres: 1
Dos: 1
Salida del hilo Uno.
Salida del hilo Dos.
Salida del hilo Tres.
Salida del hilo principal.
```

Como se puede ver, una vez que los tres hilos han comenzado, comparten el CPU. Observe la llamada a **sleep(10000)** en **main( )**, que hace que el hilo principal quede suspendido durante 10 segundos, y así se asegura que finalizará al último.

Uso de isAlive( ) y join( )

Como ya se ha mencionado, a menudo se requiere que el hilo principal sea el último en terminar. En los programas anteriores, esto se consiguió utilizando el método **sleep( )** dentro de **main( )**, con un retraso suficiente para asegurar que todos los hilos hijos terminarán antes que el hilo principal. Sin embargo, esta solución no es muy satisfactoria, y, además, sugiere una pregunta: ¿Cómo puede un hilo saber si otro ha terminado? Afortunadamente, la clase **Thread** facilita la respuesta a esta pregunta.

Existen dos formas de determinar si un hilo ha terminado. La primera consiste en llamar al método **isAlive( )** en el hilo. Éste es un método definido por la clase **Thread**, y su forma general es la siguiente:

final boolean isAlive( )

El método **isAlive( )** devuelve el valor **true** si el hilo al que se hace referencia todavía está ejecutándose, y devuelve el valor **false** en caso contrario.

El método **isAlive( )** es útil en ocasiones; sin embargo, el método, que se utiliza habitualmente para esperar a que un hilo termine es el método **join( )**. Su forma general es:

final void join( ) throws InterruptedException

Este método espera hasta que termine el hilo sobre el que se realizó la llamada. Su nombre surge de la idea de que el hilo llamante espere hasta que el hilo especificado se *reúna* con él. Otras formas de **join( )** permiten especificar un tiempo máximo de espera para que termine el hilo especificado.

A continuación se presenta una versión mejorada del ejemplo anterior que utiliza al método **join( )** para asegurar que el hilo principal es el último en terminar. También sirve como ejemplo del método **isAlive( ).**

```
// Uso de join() para esperar a que los hilos terminen.
class NewThread implements Runnable {
  String name;  // nombre del hilo
  Thread t;

  NewThread (String threadname) {
    name = threadname;
    t = new Thread(this, name);
    System.out.println ("Nuevo hilo: " + t);
    t.start(); // comienzo del hilo
  }

  // Este es el punto de entrada del hilo.
  public void run() {
    try {
      for (int i = 5; i > 0; i--) {
        System.out.println(name + ": " + i);
        Thread.sleep(l000);
      }
    } catch (InterruptedException e) {
      System.out.println ("Interrupción del hilo" + name);
    }
    System.out.println("Salida del hilo" + name);
  }
}

class DemoJoin {
  public static void main (String args[]) {
    NewThread obl = new NewThread ("Uno");
    NewThread ob2 = new NewThread ("Dos");
    NewThread ob3 = new NewThread ("Tres");
    System.out.println ("El hilo Uno está vivo: "
                        + obl.t.isAlive());
    System.out.println ("El hilo Dos está vivo: "
                        + ob2.t.isAlive());
```

```
    System.out.println ("El hilo Tres está vivo: "
                        + ob3.t.isAlive());
    //Espera a que los otros hilos terminen
  try {
    System.out.println ("Espera la finalización de los otros hilos.");
    obl.t.join ();
    ob2.t.join();
    ob3.t.join() ;
    } catch (InterruptedException e) {
      System.out.println ("Interrupción del hilo principal");
    }

    System.out.println ("El hilo Uno está vivo: "
                        + obl.t.isAlive());
    System.out.println("El hilo Dos está vivo: "
                        + ob2.t.isAlive());
    System.out.println("El hilo Tres está vivo: "
                        + ob3.t.isAlive());
    System.out.println("Salida del hilo principal.");
  }
}
```

La salida de este programa es la siguiente (la salida puede variar dependiendo de la velocidad del procesador y la carga de tareas):

```
Nuevo hilo: Thread[Uno,5,main]
Nuevo hilo: Thread[Dos,5,main]
Nuevo hilo: Thread[Tres,5,main]
El hilo Uno está vivo: true
El hilo Dos está vivo: true
El hilo Tres está vivo: true
Espera la finalización de los otros hilos.
Uno: 5
Dos: 5
Tres: 5
Uno: 4
Dos: 4
Tres: 4
Uno: 3
Dos: 3
Tres: 3
Uno: 2
Dos: 2
Tres: 2
Uno: 1
Dos: 1
Tres: 1
Salida del hilo Dos.
Salida del hilo Tres.
Salida del hilo Uno.
Hilo Uno está vivo: false
Hilo Dos está vivo: false
Hilo Tres está vivo: false
Salida del hilo principal.
```

Como se puede ver, cuando finaliza la llamada al método **join( )**, los hilos han finalizado su ejecución.

Prioridades de los Hilos

El planificador de hilos utiliza las prioridades de los hilos para decidir cuándo se debe permitir la ejecución de cada hilo. En teoría, los hilos de prioridad más alta disponen de más tiempo del CPU que los de prioridad más baja. En la práctica, el tiempo del CPU del que dispone un hilo, depende de varios factores además de su prioridad, (por ejemplo la forma en que el sistema operativo implementa la multitarea puede afectar la disponibilidad relativa de tiempo de CPU). Un hilo de prioridad alta puede desalojar a uno de prioridad más baja. Por ejemplo, cuando un hilo de prioridad más baja se está ejecutando y otro de prioridad más alta reanuda su ejecución (después de estar suspendido o esperando una E/S, por ejemplo), este segundo desalojará al de prioridad más baja.

En teoría hilos de la misma prioridad deben tener el mismo acceso al CPU, pero puede no ser exactamente así. Recuerde que Java está diseñado para funcionar en una amplia gama de entornos. Algunos de estos entornos implementan la multitarea de forma fundamentalmente diferente a otros. Por seguridad, los hilos que comparten la misma prioridad, deberían ceder el control de vez en cuando. Esto asegura que todos los hilos tienen la oportunidad de ejecutarse bajo un sistema operativo con multitarea no apropiativa. En la práctica, incluso en entornos no apropiativos, la mayoría de los hilos tienen la oportunidad de ejecutarse, ya que la mayoría de los hilos se encuentran en algún instante en una situación de bloqueo, como por ejemplo una operación de E/S. Cuando esto ocurre, el hilo que está bloqueado se suspende, y otros hilos pueden ejecutarse. Pero es mejor no confiar en esto si realmente se desea una ejecución multihilo libre de irregularidades. También hay que tener en cuenta que algunos tipos de tareas hacen un uso intensivo del CPU. Los hilos correspondientes a estas tareas dominan el CPU, y conviene que cedan el control ocasionalmente para que los otros hilos puedan ser ejecutados.

Para establecer la prioridad de un hilo se utiliza el método **setPriority( )**, que es miembro de la clase **Thread**. Su forma general es:

final void setPriority (int *nivel*)

Donde *nivel* especifica la nueva prioridad del hilo. El valor de *nivel* debe estar comprendido en el rango **MIN_PRIORITY** y **MAX_PRIORITY**. Actualmente, estos valores son 1 y 10, respectivamente. Para volver a establecer la prioridad por omisión de un hilo se utiliza el valor **NORM_PRIORITY**, que actualmente es 5. Estas prioridades están definidas como variables **final** en la clase **Thread**.

Para obtener la prioridad de un hilo se utiliza el método **getPriority( )** de **Thread**, como se indica a continuación:

final int getPriority( )

Las implementaciones de Java pueden presentar un comportamiento muy diferente en lo que a la planificación respecta. Las versiones Windows XP/98/NT/2000 funcionan, más o menos, como se podría esperar. Sin embargo, otras versiones pueden funcionar de manera absolutamente diferente. Muchas de las contradicciones surgen cuando hay hilos que adoptan un comportamiento apropiativo, en lugar de liberar el tiempo del CPU de forma cooperativa. La mejor forma para obtener un comportamiento predecible en distintas plataformas con Java es utilizar hilos que cedan el control de la CPU voluntariamente.

El siguiente ejemplo presenta dos hilos con distintas prioridades, que no se ejecutan de igual manera en plataformas apropiativas y no apropiativas. Un hilo tiene una prioridad dos niveles por encima de la prioridad normal, definida por **Thread.NORM_PRIORITY**, y el otro, dos niveles por debajo de la normal. Los dos hilos comienzan, y se permite su ejecución durante diez segundos. Cada hilo ejecuta un ciclo, contando el número de iteraciones. Después de diez segundos, el hilo principal detiene a ambos hilos y se visualiza el número de veces que cada hilo recorrió su ciclo.

```
// Ejemplo de prioridades de los hilos.
class Clicker implements Runnable {
  long click = 0;
  Thread t;
  private volatile boolean running = true;

  public clicker(int p) {
    t = new Thread(this);
    t.setPriority(p);
  }

  public void run () {
    while (running) {
      click++;
    }
  }

  public void stop () {
    running = false;
  }

  public void start () {
    t.start();
  }
}

class HiLoPri {
  public static void main (String args []) {
    Thread.currentThread().setPriority(Thread.MAX_PRIORITY) ;
    clicker hi = new clicker(Thread.NORM_PRIORITY + 2);
    clicker lo = new clicker(Thread.NORM_PRIORITY - 2);

    lo.start ();
    hi.start ();
    try {
      Thread.sleep(l0000) ;
    } catch (InterruptedException e) {
       System.out.println ("Interrupción del hilo principal.");
    }

    lo.stop();
    hi.stop();

    // Espera a que terminen los hilos hijos.
    try {
      hi.t.join();
      lo.t.join();
    } catch (InterruptedException e) {
      System.out.println ("Captura de la excepción InterruptedException");
    }
```

```
    System.out.println ("Hilo de prioridad baja: " + lo.click);
    System.out.println ("Hilo de prioridad alta: " + hi.click);
  }
}
```

Cuando este programa se ejecuta bajo Windows, la salida que se muestra a continuación, indica que los hilos realizaron el cambio de contexto, aunque ninguno de los dos cedió voluntariamente el CPU ni estuvo bloqueado por operaciones de E/S. El hilo de prioridad más alta dispuso, aproximadamente, del 90 por ciento del tiempo de CPU.

```
Hilo de prioridad baja: 4408112
Hilo de prioridad alta: 589626904
```

Obviamente, la salida exacta producida por este programa depende de la velocidad del CPU, y del número de otras tareas que se están ejecutando en el sistema. Cuando este mismo programa se ejecuta en un sistema no apropiativo, se obtienen resultados diferentes.

Otra cuestión de interés en el programa anterior es la siguiente: la variable **running** va precedido de la palabra clave **volatile**. Aunque la palabra clave **volatile** se analizará con más detalle en el capítulo 13, se utiliza aquí para asegurar que el valor de **running** será examinado cada vez que se recorra el siguiente ciclo:

```
while (running) {
click++;
}
```

Sin la utilización de la palabra clave **volatile**, Java podría optimizar el ciclo de tal forma que el valor de **running** se guardaría en una copia local. El uso de **volatile** impide esta optimización, indicando a Java que el valor de **running** puede cambiar de una manera no directamente evidente en el código inmediato.

Sincronización

Cuando dos o más hilos tienen que acceder a un recurso compartido, es necesario asegurar de alguna manera que sólo uno de ellos accede a ese recurso en cada instante. El proceso mediante el que se consigue esto se denomina *sincronización*. Como se verá, Java proporciona un soporte único, en cuanto a lenguaje, para la sincronización.

La clave para la sincronización es el concepto de monitor, también llamado *semáforo*. Un *monitor* es un objeto que se utiliza como un candado mutuamente exclusivo, o *mutex*. Sólo uno de los hilos puede poseer un monitor en un determinado instante. Cuando un hilo adquiere un candado, se dice que ha entrado en el monitor. Todos los demás hilos que intenten acceder al monitor bloqueado serán suspendidos hasta que el primero salga del monitor. Se dice que estos otros hilos están *esperando* al monitor. Un hilo que posea un monitor puede volver a entrar en el mismo monitor si así lo desea.

Si ha trabajado con sincronización al utilizar otros lenguajes, como C y C++, sabrá que puede resultar un tanto compleja. Esto se debe a que la mayoría de lenguajes no implementan la sincronización, sino que, para la sincronización de hilos, utilizan funciones primitivas del sistema operativo. Afortunadamente, Java implementa la sincronización mediante elementos del lenguaje, con lo que la mayor parte de la complejidad asociada a la misma ha sido eliminada.

Un código se puede sincronizar de dos formas. Ambas implican el uso de la palabra clave **synchronized**, y se analizan a continuación.

Métodos sincronizados

La sincronización resulta sencilla en Java, porque todos los objetos tienen su propio monitor implícito asociado. Para entrar en el monitor de un objeto, basta con llamar a un método modificado con la palabra clave **synchronized**. Mientras un hilo esté dentro de un método sincronizado, todos los demás hilos que traten de llamar a ese método, o a otro método sincronizado sobre la misma instancia, tendrán que esperar. Para salir del monitor y abandonar el control del objeto, el propietario del monitor sólo tiene que salir del método sincronizado.

Para entender mejor que la sincronización es necesaria, comencemos con un ejemplo sencillo que no la usa, pero debería. El siguiente programa tiene tres clases. La primera, **Callme**, tiene un sólo método llamado **call( )**. El método **call( )** tiene un parámetro del tipo **String** denominado **msg**. Este método intenta imprimir la cadena **msg** entre corchetes. La cuestión de interés es que, después de que el método **call( )** imprime el corchete de apertura y la cadena **msg**, se llama a **Thread.sleep(l000)**, lo que detiene el hilo en curso durante un segundo.

El constructor de la siguiente clase, **Caller**, toma una referencia a una instancia de la clase **Callme** y un **String**, los cuales se almacenan en las variables **target** y **msg** respectivamente. El constructor también crea un nuevo hilo que llamará al método **run** de este objeto. El hilo es iniciado inmediatamente. El método **run( )** de **Caller** llama al método **call( )** de la instancia **target** de **Callme**, pasando la cadena **msg**. Finalmente, la clase **Synch** comienza creando una instancia de **Callme**, y tres instancias de **Caller**, cada una con una cadena diferente. La misma instancia de **Callme** se pasa a cada instancia de **Caller**.

```
// Este programa no está sincronizado.
class Callme {
  void call (String msg) {
    System.out.print ("[" + msg);
    try {
      Thread.sleep (l000);
    } catch (InterruptedException e) {
      System.out.println ("Interrumpido");
    }
    System.out.println ("]");
  }
}

class Caller implements Runnable {
  String msg;
  Callme target;
  Thread t;
  public Caller (Callme targ, String s) {
    target = targ;
    msg = s;
    t = new Thread(this);
    t.start() ;
  }

    public void run() {
      target.call(msg);
    }
  }
```

```
  class Synch {
    public static void main (String args[]) {
      Callme target = new Callme();
      Caller obl = new Caller (target, "Hola");
      Caller ob2 = new Caller (target, "Sincronizado");
      Caller ob3 = new Caller (target, "Mundo".) ;

      // espera a que terminen los hilos
    try {
      obl.t.join();
      ob2.t.join();
      ob3.t.join();
    } catch (InterruptedException e) {
      System.out.println ("Interrumpido");
    }
  }
}
```

La salida producida por este programa es la siguiente:

```
[Hola [Sincronizado [Mundo]
]
]
```

Al llamar a **sleep( )**, el método **call( )** permite cambiar la ejecución a otro hilo. El resultado es una salida en la que se mezclan los tres mensajes de forma confusa. En este programa no hay nada que impida a los tres hilos llamar al mismo método, en el mismo objeto y al mismo tiempo. Esto es lo que se conoce como una condición de carrera (*race condition*), ya que los tres métodos compiten uno con otro para completar el método. Este ejemplo utiliza **sleep( )** para que los efectos sean repetibles y obvios. En la mayoría de las situaciones, una condición de carrera es más sutil y menos predecible, porque no se puede tener seguridad de cuándo se produce el cambio de contexto. Esto puede dar lugar a que un programa se ejecute de forma correcta unas veces e incorrecta otras.

Para corregir el programa anterior, se debe producir un acceso en *serie* al método **call( )**, es decir, se debe restringir el acceso a un único hilo en cada instante. Para ello, simplemente hay que colocar por delante de la definición del método **call( )** la palabra clave **synchronized**, tal y como se muestra a continuación:

```
class Callme {
     synchronized void call (String msg) {
     ...
```

Esto impide que otros hilos accedan al método **call( )** mientras un determinado hilo lo está utilizando. Después de añadir la palabra **synchronized** al método **call( )**, la salida del programa es la siguiente:

```
[Hola]
[Sincronizado]
[Mundo]
```

Siempre que se tenga un método, o un grupo de métodos, que manipulan el estado interno de un objeto en una situación de múltiples hilos, se debe usar la palabra clave **synchronized** para salvaguardar dicho estado de las condiciones de carrera. Recuerde que una vez que un hilo

entra en un método sincronizado de una instancia, ningún otro hilo puede entrar en ningún otro método sincronizado de la misma instancia. Sin embargo, sí se podrá llamar a métodos no sincronizados de la misma instancia.

La sentencia synchronized

La creación de métodos **sincronizados** en clases creadas por el programador es una forma fácil y efectiva de conseguir la sincronización; sin embargo, no funciona con todas las clases. Veamos por qué. Suponga que quiere sincronizar el acceso a objetos de una clase que no fue diseñada para el acceso de múltiples hilos, es decir, la clase no utiliza métodos **sincronizados**. Además, la clase fue creada por otros programadores, y no tiene acceso al código fuente. Por lo tanto, no puede añadir la palabra clave **synchronized** a los métodos necesarios. ¿Cómo se puede conseguir que el acceso a un objeto de esa clase sea sincronizado? Afortunadamente, la solución es fácil. Simplemente hay que poner llamadas a los métodos definidos por esa clase dentro de un bloque sincronizado.

Ésta es la forma general de la sentencia **synchronized**:

```
synchronized (objeto) {
  // sentencias que deben ser sincronizadas
  }
```

donde *objeto* es una referencia al objeto que se quiere sincronizar. Si se quiere sincronizar una única sentencia, no son necesarias las llaves. Un bloque sincronizado asegura que sólo se producirá una llamada a un método miembro de *objeto* después de que el hilo actual haya entrado en el monitor del *objeto*.

La siguiente es una versión alternativa del ejemplo anterior, que utiliza un bloque sincronizado dentro del método **run( )**:

```
// Este programa utiliza un bloque sincronizado.
class Callme {
  void call (String msg) {
    System.out.print ("[" + msg);
    try {
      Thread.sleep (l000);
    } catch (InterruptedException e) {
      System.out.println ("Interrumpido");
    }
    System.out.println ("]");
  }
}

class Caller implements Runnable {
  String msg;

  Callme target;
  Thread t;

  public Caller (Callme targ, String s) {
    target = targ;
    msg = s;
    t = new Thread (this);
    t.start();
  }
```

```
  // Sincronización de las llamadas a call()
  public void run() {
    synchronized (target) { // Bloque sincronizado
      target.call (msg);
    }
  }
}

class Synchl {
  public static void main (String args[]) {
    Callme target = new Callme();
    Caller obl = new Caller (target, "Hola") ;
    Caller ob2 = new Caller (target, "Sincronizado");
    Caller ob3 = new Caller (target, "Mundo");

    // Espera a que los hilos terminen
    try {
      obl.t.join();
      ob2.t.join();
      ob3.t.joinO;
    } catch(InterruptedException e) {
      System.out.println ("Interrumpido");
    }
  }
}
```

Aquí no se ha modificado con la palabra **synchronized** al método **call( )**. En su lugar, se utiliza la sentencia **synchronized** dentro del método **run( )** de **Caller**. La salida que se obtiene es la misma que en el ejemplo anterior, ya que cada hilo espera a que el anterior termine antes de proceder.

Comunicación entre hilos

En los ejemplos anteriores se bloqueaba el acceso asíncrono a ciertos métodos para los demás hilos. Este uso de los monitores implícitos de los objetos en Java es bastante eficaz, pero se puede conseguir un nivel más refinado de control mediante la comunicación entre procesos, la cual es especialmente simple en Java.

Como se ha explicado anteriormente, la programación multihilo sustituye la programación basada en ciclos de eventos, al dividir las tareas en unidades discretas y lógicas. Los hilos tienen, además, una segunda ventaja: permiten eliminar el sondeo, que es un mecanismo mediante el cual se comprueba de forma repetitiva si se cumple una condición. Cuando dicha condición se cumple, se ejecuta una determinada acción. Esto supone un desaprovechamiento del CPU.

Consideremos, por ejemplo, el problema clásico de colas, en que un hilo está produciendo unos datos y otro los está consumiendo. Para hacer el problema más interesante, consideremos, además, que el hilo productor tiene que esperar hasta que el hilo consumidor termine, antes de generar más datos.

En un sistema con sondeo, el hilo consumidor desperdiciaría muchos ciclos de CPU esperando la producción de datos. Una vez que el productor hubiera finalizado, comenzaría el sondeo, desaprovechándose más ciclos de CPU hasta que el hilo consumidor terminara, etc. Esta situación evidentemente no es deseable.

Para evitar el sondeo, Java aporta un elegante mecanismo de comunicación entre procesos por medio de los métodos **wait( )**, **notify( )** y **notifyAll( )**. Estos métodos se han implementado como métodos **final** en la clase **Object**, de forma que están incluidos en todas las clases automáticamente. Sólo se puede llamar a estos tres métodos desde dentro de un método **sincronizado**. Las reglas de uso de estos tres métodos son bastante sencillas, aunque conceptualmente avanzadas desde la perspectiva de las ciencias de la computación:

- **wait( )** indica al hilo que realiza la llamada que debe abandonar el monitor y quedar suspendido hasta que algún otro hilo entre en el mismo monitor y llame al método **notify( ).**
- **notify( )** activa un hilo que previamente llamó a **wait( )** en el mismo objeto.
- **notifyAll( )** activa todos los hilos que llamaron previamente a **wait( )** en el mismo objeto. Uno de esos hilos comenzará a ejecutarse.

Estos métodos se declaran dentro de la clase **Object**, tal y como se muestra a continuación:

```
final void wait( ) throws InterruptedException
final void notify( )
final void notifyAll( )
```

Existen formas adicionales de **wait( )** que permiten especificar un determinado periodo de espera.

Antes de pasar a un ejemplo que ilustre la comunicación entre los hilos, es importante tratar otro aspecto. Aunque **wait( )** normalmente espera hasta que **notify( )** o **notifyAll( )** sea llamado, existe una posibilidad que en casos muy raros el hilo en espera pueda ser despertado debido a una *falsa alarma*. En este caso, un hilo de espera puede reiniciar sin que **notify( )** o **notifyAll( )** hayan sido llamadas, en esencia el hilo reinicia sin razón aparente. Dada esta remota posibilidad, la empresa SUN (creadora de Java) recomienda que las llamadas a **wait( )** se realicen dentro de un ciclo que compruebe la condición de los hilos que están esperando. El siguiente ejemplo muestra esta técnica.

Veremos a continuación un ejemplo que usa **wait( )** y **notify( )**. Para comenzar considere el siguiente ejemplo de programa que implementa de manera incorrecta una forma sencilla del problema de productor/consumidor. El ejemplo consiste en cuatro clases: **Q**, la cola que se intenta sincronizar; **Producer**, el objeto hilo que genera los datos para la cola; **Consumer**, el hilo objeto que consume los datos de la cola, y **PC**, la mini clase que crea las clases **Q**, **Producer** y **Consumer**.

```
// Una implementación incorrecta del problema de productor / consumidor.
class Q {
  int n;

  synchronized int get() {
    System.out.println ("Consume: " + n);
    return n;
  }

  synchronized void put (int n) {
    this.n = n;
    System.out.println ("Produce: " + n);
  }
}
```

```
class Producer implements Runnable {
  Q q;

  Producer(Q q) {
    this.q = q;
    new Thread (this, "Productor").start();
  }

  public void run(){
    int i = 0;

    while (true) {
      q.put (i++) ;
    }
  }
}

class Consumer implements Runnable {
  Q q;

  Consumer (Q q) {
    this.q = q;
    new Thread(this, "Consumidor").start();
  }

  public void run() {
    while(true) {
      q.get() ;
    }
  }
}

class PC {
  public static void main (String args[]) {
    Q q = new Q();
    new Producer(q);
    new Consumer(q);

    System.out.println ("Pulse Control-C para finalizar.");
  }
}
```

Aunque los métodos **put( )** y **get( )** de **Q** son métodos sincronizados, nada impide que el productor vaya más rápido que el consumidor, ni que el consumidor recolecté el mismo valor de la cola dos veces. Por ello, se obtienen las salidas que se muestran continuación, visiblemente incorrectas. La salida exacta depende de la velocidad del procesador y de la carga de tareas.

```
Produce: 1
Consume: 1
Consume: 1
Consume: 1
Consume: 1
Consume: 1
```

```
    Produce: 2
    Produce: 3
    Produce: 4
    Produce: 5
    Produce: 6
    Produce: 7
    Consume: 7
```

Como se puede ver, después de que el productor genera un 1, el consumidor comienza y obtiene el mismo 1 cinco veces seguidas. Entonces, el productor continúa y genera los valores del 2 al 7, sin dejar al consumidor la oportunidad de obtenerlos.

La forma correcta de escribir este programa en Java consiste en utilizar los métodos **wait( )** y **notify( )** para la comunicación en ambos sentidos:

```
// una implementación correcta del problema productor / consumidor.
class Q {
  int n;
  boolean valueSet = false;

  synchronized int get() {
    while (!valueSet)
      try {
        wait () ;

      } catch (InterruptedException e) {
         System.out.println ("Captura de la excepción InterruptedException");
       }

       System.out.println ("Consume: " + n);
       valueSet = false;
       notify () ;
       return n;
}

synchronized void put (int n) {
  while (valueSet)
    try {
      wait ();
    } catch(InterruptedException e) {
      System.out.println ("Captura de la excepción de InterruptedException");
    }

    this.n = n;
    valueSet = true;
    System.out.println ("Produce: " + n);
    notify();
  }
}

class Producer implements Runnable {
  Q q;

  Producer (Q q) {
    this.q = q;
    new Thread (this, "Productor").start();
  }
```

```
  public void run(){
    int i = 0;

    while (true) {
      q.put (i++);
    }
  }
}

class Consumer implements Runnable {
  Q q;

  Consumer (Q q) {
    this.q = q;
    new Thread (this, "Consumidor").start();
  }

  public void run()
    while (true) {
      q.get();
    }
  }
}

class PCFixed {
  public static void main (String args[]){
    Q q = new Q();
    new Producer(q);
    new Consumer(q);

    System.out.println ("Pulse Control+C para finalizar.");
  }
}
```

Dentro de **get( )**, se llama a **wait( )**. Esto ocasiona que se suspenda la ejecución hasta que **Producer** notifique que los datos están listos. Cuando esto sucede, se reanuda la ejecución dentro de **get( )**. Una vez obtenidos los datos, desde el método **get( )** se llama a **notify( )**. Esto indica a **Producer** que puede colocar más datos en la cola. Dentro de **put( )**, el método **wait( )** suspende la ejecución hasta que **Consumer** haya retirado el dato de la cola. Cuando la ejecución continúa, se coloca el siguiente dato en la cola y se llama a **notify( )**, lo que indica a **Consumer** que debe retirarlo.

La salida generada muestra el comportamiento correcto:

```
Produce:   1
Consume:   1
Produce:   2
Consume:   2
Produce:   3
Consume:   3
Produce:   4
Consume:   4
Produce:   5
Consume:   5
```

Bloqueos

Un tipo especial de error, que es necesario evitar y está relacionado específicamente con la multitarea, es el bloqueo (mejor conocido como *deadlock* por su nombre en inglés). Este error se produce cuando dos hilos tienen una dependencia circular en una pareja de objetos sincronizados. Supongamos, por ejemplo, que un hilo entra en el monitor sobre el objeto X y otro hilo en el monitor sobre el objeto Y. Si el hilo de X intenta llamar a cualquier método sincronizado del objeto Y, tal y como se puede esperar, quedará bloqueado. Sin embargo, si el hilo de Y, a su vez, intenta llamar a cualquier método sincronizado de X, quedará esperando indefinidamente, ya que, para acceder a X, tendrá que liberar antes su propio candado en Y con objeto de que el primer hilo pudiera finalizar. El bloqueo es un error difícil de depurar, por dos razones:

- En general, ocurre pocas veces cuando los dos hilos coinciden en el tiempo de forma correcta.
- Puede implicar a más de dos hilos y dos objetos sincronizados, es decir, el bloqueo puede darse en una secuencia de eventos más compleja que la que se acaba de describir.

Para una comprensión completa del bloqueo, es útil ver cómo se produce en la práctica. En el siguiente ejemplo se crean dos clases, **A** y **B**, con los métodos **foo( )** y **bar( )**, respectivamente, que hacen una breve pausa cada uno antes de llamar al método de la otra clase. La clase principal, denominada **Deadlock**, crea una instancia de **A** y otra de **B**, dando lugar a un segundo hilo para establecer la condición de bloqueo. Los métodos **foo( )** y **bar( )** utilizan **sleep( )** para obligar a que se produzca la condición de bloqueo.

```
// Un ejemplo de bloqueo.
class A {
  synchronized void foo(B b) {
    String name = Thread.currentThread().getName();

    System.out.println (name + "entra en A.foo");

    try{
      Thread.sleep (l000);
    } catch (Exception e) {
      System.out.println ("Se interrumpe A");
    }

    System.out.println (name + " intenta llamar al método B.last()");
    b.last();
  }

  synchronized void last() {
    System.out.println ("Dentro de A.last");
  }
}

class B {

synchronized void bar (A a) {
  String name = Thread.currentThread().getName();
  System.out.println (name + " entra en B.bar") ;
```

```
  try {
    Thread.sleep(l000);
    } catch(Exception e) {
      System.out.println ("Se interrumpe B");
    }

    System.out.println (name + " intenta llamar a A.last()");
    a.last ();
  }

  synchronized void last() {
    System.out.println ("Dentro de A.last");
  }
}

class Deadlock implements Runnable {
  A a = new A();
  B b = new B();

  Deadlock() {
    Thread.currentThread().setName("Hilo Principal");
    Thread t = new Thread(this, "Hilo hijo");
    t.start();

    a.foo(b); // Este hilo se bloquea en a.
    System.out.println ("Regresa al hilo principal");
  }

  public void run() {
    b.bar(a); // Este hilo se bloquea en b.
    System.out.println ("Regresa al otro hilo");
  }

  public static void main (String args[]){
    new Deadlock () ;
  }
}
```

Al ejecutar este programa se obtiene la siguiente salida:

```
Hilo principal entra en A.foo
Hilo hijo entra en B.bar
Hilo principal intenta llamar a B.last()
Hilo hijo intenta llamar a A.last()
```

Al ejecutar el programa, el sistema se bloquea, por ello es necesario presionar CTRL+C para finalizar. El volcado completo del hilo y de la memoria caché completos se puede ver presionando CTRL+BREAK en una PC. De esta forma se comprueba que el **Hilo hijo** posee el monitor de **b** mientras está esperando el monitor de **a**.

Al mismo tiempo, el **Hilo principal** posee a **a** y está esperando obtener **b**. Este programa no se completará nunca. Como lo ilustra este ejemplo, si un programa multihilo no funciona correctamente, una de las primeras condiciones que se deben revisar es el bloqueo.

Suspensión, reanudación y finalización de hilos

Algunas veces es necesario suspender la ejecución de un hilo. Por ejemplo, un hilo se puede utilizar para visualizar la hora del día, si el usuario no quiere este reloj, se puede suspender a este hilo. Cualquiera que sea el caso, suspender un hilo es sencillo, y, una vez suspendido, volverlo a activar también es fácil.

Los mecanismos que se utilizan en las nuevas versiones de Java, a partir de Java 2, para suspender, finalizar y reanudar un hilo, son diferentes a los existentes en las versiones previas. Aunque para cualquier nuevo código se debe utilizar el enfoque de Java 2, es conveniente comprender cómo se realizaban estas operaciones en entornos con las versiones anteriores, si se quiere actualizar o mantener un código antiguo. También es necesario comprender el motivo de los cambios que se realizan en Java 2. Por estas razones, la siguiente sección describe la forma original en que se controlaba la ejecución de un hilo, y en una sección posterior se describe el enfoque empleado en Java 2.

Suspensión, reanudación y finalización de hilos con Java 1.1 y versiones anteriores

Antes de Java 2, un programa utilizaba los métodos **suspend( )** y **resume( )**, definidos por la clase **Thread**, para parar y reanudar la ejecución de un hilo. La forma general de estos métodos es:

```
final void suspend( )
final void resume( )
```

El siguiente programa es un ejemplo del uso de estos métodos:

```
// Uso de suspend() y resume().
class NewThread implements Runnable {
  String name;  // nombre del hilo
  Thread t;

  NewThread (String threadname) {
    name = threadname;
    t = new Thread (this, name);
    System.out.println ("Nuevo hilo: " + t);
    t.start(); // Comienzo del hilo
  }

  // Este es el punto de entrada del hilo.
  public void run() {
    try {
      for (int i = 15; i > 0; i--) {
        System.out.println (name + ": " + i);
          Thread.sleep(200);
      }
    } catch (InterruptedException e) {
      System.out.println (" Interrupción del hilo" + name);
    }
    System.out.println (" Salida del hilo" + name);
  }
}
```

```
class SuspendResume {
  public static void main (String args[]) {
    NewThread obl = new NewThread ("Uno");
    NewThread ob2 = new NewThread ("Dos");

    try {
      Thread.sleep(l000);
      obl.t.suspend() ;
      System.out.println ("Suspensión del hilo Uno");
      Thread.sleep(l000);
      obl.t.resume() ;
      System.out.println ("Reanudación del hilo Uno");
      ob2.t.suspend() ;
      System.out.println ("Suspensión del hilo Dos");
      Thread.sleep(l000);
      ob2.t.resume();
      System.out.println ("Reanudación del hilo Dos");
    } catch (InterruptedException e) {
      System.out.println ("Interrupción del hilo principal");
    }

    // Espera a que terminen los otros hilos
    try {
      System.out.println ("Espera la finalización de los otros hilos.");
      ob1.t .join();
      ob2.t.join() ;
    } catch (InterruptedException e) {
      System.out.println ("Interrupción del hilo principal");
    }
    System.out.println ("Salida del hilo principal.");
  }
}
```

La salida generada por este programa es la siguiente (la salida puede variar por la velocidad del procesador y la carga de tareas).

```
Nuevo hilo: Thread[Uno,5,main]
Uno: 15
Nuevo hilo: Thread[Dos,5,main]
Dos: 15
Uno: 14
Dos: 14
Uno: 13
Dos: 13
Uno: 12
Dos: 12
Uno: 11
Dos: 11
Suspensión del hilo Uno
Dos: 10
Dos: 9
Dos: 8
```

```
Dos: 7
Dos: 6
Reanudación del hilo Uno
Suspensión del hilo Dos
Uno: 10
Uno: 9
Uno: 8
Uno: 7
Uno: 6
Reanudación del hilo Dos
Espera la finalización de los otros hilos.
Dos: 5
Uno: 5
Dos: 4
Uno: 4
Dos: 3
Uno: 3
Dos: 2
Uno: 2
Dos: 1
Uno: 1
Salida del hilo Dos.
Salida del hilo Uno.
Salida del hilo principal.
```

La clase **Thread** también define un método, llamado **stop( )**, que finaliza el hilo. Su forma general es:

final void stop( )

Una vez finalizado, un hilo no puede reanudarse utilizando el método **resume( )**.

La forma moderna de suspensión, reanudación y finalización de hilos

Aunque los métodos **suspend( )**, **resume( )** y **stop( )**, definidos por la clase **Thread**, parecen razonables y un enfoque adecuado para la gestión de la ejecución de los hilos, no deben ser utilizados por los nuevos programas de Java. La razón es la siguiente. El método **suspend( )** de la clase **Thread** ha sido descontinuado en Java 2 debido a que puede dar lugar a fallos graves del sistema. Suponiendo que un hilo ha obtenido el acceso exclusivo sobre estructuras de datos críticos, si ese hilo se suspende, no abandona ese acceso exclusivo. Por ende, otros hilos que pueden estar esperando esos recursos podrían estar bloqueados.

También se descontinúa el método **resume( )**, ya que aunque no causa problemas no se puede usar sin su equivalente método **suspend( )**.

El método **stop( )** de la clase **Thread** también se descontinúa en Java 2, debido a que también puede causar graves fallos del sistema. Supongamos que un hilo está escribiendo en una estructura de datos importante y que sólo ha completado parte de los cambios. Si ese hilo se finaliza en ese momento, esa estructura de datos podría quedar en un estado corrupto.

Al no poder usar los métodos **suspend( )**, **resume( )** o **stop( )** en Java 2 para controlar un hilo, se podría pensar que no hay forma de parar, reiniciar o terminar un hilo, pero afortunadamente esto no es así. Un hilo debe ser diseñado de forma que el método **run( )**

compruebe periódicamente si ese hilo debe suspender, reanudar o finalizar su propia ejecución. Normalmente esto se realiza estableciendo una variable bandera que indica el estado de la ejecución del hilo. Mientras esta variable tenga asignado el valor "ejecutar", el método **run( )** debe continuar dejando que el hilo se ejecute. Si se asigna a esta variable el valor "suspender", el hilo debe parar, y si se le asigna el valor "finalizar", el hilo debe terminar. Naturalmente, existen muchas formas diferentes en las que se puede escribir el código correspondiente, pero la idea es la misma para todos los programas.

El siguiente ejemplo pone de manifiesto cómo se pueden utilizar los métodos **wait( )** y **notify( )**, heredados de **Object**, para controlar la ejecución de un hilo. Este ejemplo es semejante al de la sección anterior; sin embargo se han eliminado las llamadas a los métodos descontinuados. Veamos cómo funciona este programa.

La clase **NewThread** contiene una variable de instancia **boolean** denominada **suspendFlag**, que se utiliza para controlar la ejecución del hilo. El constructor inicializa a **suspendFlag** con el valor **false**. El método **run( )** contiene una sentencia de bloque **synchronized** que revisa la variable **suspendFlag**. Si esa variable tiene el valor **true**, se invoca al método **wait( )** para suspender la ejecución del hilo. El método **mysuspend( )** asigna a la variable **suspendFlag** el valor **true**. El método **myresume( )** asigna a la variable **suspendFlag** el valor **false** e invoca a **notify( )** para reactivar el hilo. Finalmente, se ha modificado el método **main( )** para llamar a los métodos **mysuspend( )** y **myresume( ).**

```
// Versión moderna de suspensión y reanudación de un hilo
class NewThread implements Runnable {
  String name;  // nombre del hilo
  Thread t;
  boolean suspendFlag;

  NewThread (String threadname) {
    name = threadname;
    t = new Thread(this, name);
    System.out.println ("Nuevo hilo: " + t);
    suspendFlag = false;
    t.start(); // Comienzo del hilo
  }

  // Este es el punto de entrada del hilo.
  public void run() {
    try {
      for (int i = 15; i > 0; i--) {
        System.out.println (name + ": " + i);
        Thread.sleep (200);
        synchronized (this) {
        while (suspendFlag) {
        wait() ;
       }
    }
  }
} catch (InterruptedException e) {
  System.out.println ("Interrupción del hilo" + name);
}
```

```
    System.out.println ("Salida del hilo" + name);
  }

  void mysuspend () {
    suspendFlag = true;
  }

  synchronized void myresume()
    suspendFlag = false;
    notify ();
  }
}

class SuspendResume {
  public static void main (String args[]) {
    NewThread obl = new NewThread("Uno");
    NewThread ob2 = new NewThread("Dos") ;

    try {
      Thread.sleep (l000);
      obl.mysuspend ();
      System.out.println ("Suspensión del hilo Uno");
      Thread.sleep (l000);
      obl.myresume();
      System.out.println ("Reanudación del hilo Uno");
      ob2.mysuspend() ;
      System.out.println ("Suspensión del hilo Dos");
      Thread.sleep(l000);
      ob2.myresume();
      System.out.println ("Reanudación del hilo Dos");
    } catch (InterruptedException e) {
      System.out.println ("Interrupción del hilo principal");
    }

    // espera a que los otros hilos terminen
    try {
      System.out.println ("Espera la finalización de los otros hilos.");
      obl.t.join () ;
      ob2.t.join();
    } catch (InterruptedException e) {
      System.out.println ("Interrupción del hilo principal");
    }

    System.out.println ("Salida del hilo principal.");
  }
}
```

La salida de este programa es la misma que la que aparece en el apartado anterior. Más adelante se verán más ejemplos en los que se usa el mecanismo moderno de control de hilos. Aunque este mecanismo no es tan claro como el de la versión anterior, es la forma de asegurar que no se producirán errores en tiempo de ejecución, y es el enfoque que se *debe* utilizar en el nuevo código.

Programación multihilo

La clave para utilizar de manera eficaz las características multihilo de Java es pensar de manera concurrente, en lugar de hacerlo de forma lineal o en serie. Por ejemplo, si se tienen dos subsistemas de un programa que se pueden ejecutar concurrentemente, conviene hacer, con cada uno de esos subsistemas, hilos individuales. Con un uso adecuado de la programación multihilo se pueden crear programas muy eficientes. Sin embargo, conviene tener la precaución de no crear demasiados hilos, ya que en ese caso se puede degradar el rendimiento del programa en lugar de mejorarlo. Conviene recordar que el cambio de contexto lleva asociado una carga de trabajo adicional. Si se crean demasiados hilos, se gastará más tiempo de CPU en los cambios de contexto entre hilos que en la ejecución del programa.

CAPÍTULO 12

Enumeraciones, autoboxing y anotaciones (metadatos)

Este capítulo examina tres anexos recientes en el lenguaje Java: enumeraciones, autoboxing y anotaciones (también llamadas metadatos). Cada uno de ellos extiende el poder del lenguaje al ofrecer una forma estilizada de gestionar tareas comunes de programación. Este capítulo también presenta los tipos envueltos de Java e introduce el concepto de reflexión.

Enumeraciones

Versiones anteriores a JDK 5 carecían de una característica que muchos programadores sentían era necesaria: enumeraciones. En su forma simple, una *enumeración* es una lista de constantes. Aunque Java ofrece otras características que proveen de alguna manera funcionalidades similares, tales como las variables **final**, para muchos programadores aún faltaba el concepto puro de enumeraciones –especialmente porque las enumeraciones están presentes en la mayoría de los lenguajes comúnmente utilizados. A partir de JDK 5, las enumeraciones fueron agregadas al lenguaje Java, y ahora están disponibles para los programadores en Java.

En la forma más simple, las numeraciones en Java parecen similares a las enumeraciones de otros lenguajes. Sin embargo, esta similitud es sólo superficial. En lenguajes como C++, las enumeraciones simplemente son listas de constantes de tipo entero. En Java, una enumeración define un tipo (una clase), esto expande enormemente el concepto de enumeración. Por ejemplo, en Java, una enumeración puede tener constructores, métodos y variables de instancia. Además, aunque las enumeraciones en Java tardaron varios años en aparecer, la rica implementación hecha de ellas en Java justifica la espera.

Fundamentos de las enumeraciones

Una enumeración se crea utilizando la palabra clave **enum**. Por ejemplo, ésta es una enumeración simple que lista algunas categorías de manzanas.

```
//Una enumeración de categorías de manzanas
enum Manzana {
  Jonathan, GoldenDel, RedDel, Winesap, Cortland
}
```

Nota de los traductores: Hemos preferido dejar la palabra autoboxing sin traducir. El término hace referencia al proceso de convertir un dato primitivo en un objeto equivalente automáticamente. Se dice que el dato original es colocado dentro del objeto, como un regalo dentro de una caja.

Los identificadores **Jonathan**, **GoldenDel** y el resto, son llamados *constantes de enumeración*. Cada uno está implícitamente declarado como un miembro de tipo **public**, **static** y **final** de la clase **Manzana**. Además, su tipo es el tipo de la enumeración en la cual fueron declarados, en este caso es **Manzana**.

Una vez que se tiene definida una enumeración, se puede crear una variable de ese tipo. Sin embargo, aunque las enumeraciones definen a una clase tipo, no se instancia un **enum** usando **new**. En lugar de eso, se declara y usa una variable enumeración tal como se hace con los tipos primitivos. Por ejemplo, el siguiente código declara **ap** como una variable del tipo enumerado **Manzana**:

```
Manzana ap;
```

Dado que **ap** es de tipo **Manzana**, los únicos valores que le pueden ser asignados (o puede contener) son aquellos definidos por la enumeración. Por ejemplo, la siguiente línea asigna a **ap** el valor **RedDel:**

```
ap = Manzana.RedDel;
```

Note que el símbolo **RedDel** es precedido por **Manzana**.

Dos constantes de enumeración pueden ser comparadas en busca de una igualdad utilizando el operador relacional ==. Por ejemplo, la siguiente sentencia compara el valor de **ap** con el de la constante **GoldenDel:**

```
if (ap == Manzana.GoldenDel) //…
```

Un valor de enumeración puede también ser utilizado para controlar una sentencia **switch**. Claro está que todas las sentencias **case** deben ser constantes de la misma variable enumerada utilizada en la expresión de **switch**. Por ejemplo, la siguiente es una sentencia **switch** perfectamente válida:

```
//Usa una enumeración para controlar una sentencia switch
switch (ap) {
  case Jonathan;
    // …
  case Winesap;
    // …
```

Note que las sentencias **case**, los nombres de las constantes enumeradas son listadas sin estar precedidas por el nombre de sus tipo de enumeración. Esto es, **Winesap** se utiliza en lugar de **Manzana.Winesap**. Esto se debe a que el tipo de la enumeración de la variable en la expresión **switch** específica implícitamente el tipo **enumerado** para las constantes utilizadas en las sentencias **case**. No es necesario utilizar el nombre de la **enumeración** junto al nombre de las constantes en la sentencia **case**. De hecho, hacerlo causaría un error de compilación.

Cuando una constante enumerada es mostrada en pantalla con una sentencia **println( )**, su nombre es mostrado en pantalla. Por ejemplo, la siguiente sentencia

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

Despliega en pantalla el nombre **Winesap**.

El siguiente programa coloca todas las piezas juntas utilizando la enumeración **Manzana**:

```
// Una enumeración de tipos de manzanas
enum Manzana {
  Jonathan, GoldenDel, RedDel, Winesap, Cortland
}

class EnumDemo {
  public static void main(String args[])
  {
    Manzana ap;

    ap = Apple.RedDel;

    // mostrar en pantalla un valor de tipo enum
    System.out.println("Valor de ap: "+ ap);
    System.out.println();

    ap = Manzana.GoldenDel;

    // comparar dos valores de tipo enum
    if(ap == Apple.GoldenDel)
      System.out.println("ap contiene GoldenDel.\n");

    // uso de una variable enum en una sentencia switch
    switch (ap) {
      case Jonathan:
        System.out.println ("La manzana Jonathan es roja.");
        break;
      case GoldenDel:
        System.out.println("La manzana Golden Delicious es amarilla.");
        break;
      case RedDel:
        System.out.println("La manzana Red Delicious es roja.");
        break;
      case Winesap:
        System.out.println("La manzana Winesap es roja.");
        break;
      case Cortland:
        System.out.println("La manzana Cortland es roja.");
        break;
    }
  }
}
```

La salida de este programa se muestra a continuación:

```
El valor de ap: RedDel

ap contiene: GoldenDel.

La manzana Golden Delicious es amarilla.
```

Los métodos values( ) y valuesOf( )

Todas las enumeraciones automáticamente contienen dos métodos predefinidos: **values( )** y **valueOf( )**. La siguiente es su forma general:

public static *enum-type*[ ] values( )
public static *enum-type* valueOf(String *str*)

El método **values( )** regresa un arreglo que contiene una lista de constantes enumeradas. El método **valueOf( )** regresa la constante enumerada cuyo valor corresponde a la cadena pasada en el parámetro *str*. En ambos casos, *enum-type* es el tipo de enumeración. Por ejemplo, en el caso de la enumeración **Manzana** que se mostró anteriormente, **Manzana.valueOf("Winesap")** regresa **Winesap**.

El siguiente programa muestra el uso de los métodos **values( )** y **valueOf( )**:

```
// Uso de los métodos predefinidos para las enumeraciones

// Una enumeración de tipos de manzana.
enum Manzana {
  Jonathan, GoldenDel, RedDel, Winesap, Cortland
}

class EnumDemo2 {
  public static void main(String args[])
  {
    Manzana ap;

    System.out.println("Estas son todas las constantes de tipo Manzana:");

    // usando el método values()
    Manzana allapples[] = Manzana.values();
    for(Manzana a : allapples)
      System.out.println(a) ;

    System.out.println();

    // usando el método valueOf ()
    ap = Manzana.valueOf ("Winesap") ;
    System.out.println("ap contiene " + ap);

  }
}
```

La salida del programa es la siguiente:

```
Estas son todas las constantes de tipo Manzana:
Jonathan
Golden Del
RedDel
Winesap
Cortland

ap contiene Winesap
```

Nótese que el programa utiliza un ciclo estilo for-each el cual itera a través del arreglo de constantes obtenidas cuando se llama al método **values( )**. Para ilustrar esto, se creó la variable **allapples** y se le asignó una referencia a un arreglo con los valores de la enumeración. Sin embargo, este paso no es necesario porque el **for** podría haber sido escrito como se muestra a continuación, eliminando la necesidad de la variable **allapples**:

```
for (Manzana a: Manzana.values())
  System.out.println(a);
```

Ahora, nótese cómo el valor correspondiente al nombre **Winesap** fue obtenido por la llamada al método **valueOf( )**.

```
ap = Manzana.valueOf("Winesap");
```

Como se explicó antes, **valueOf( )** regresa el valor en la enumeración asociado con el nombre de la constante representada como una cadena.

NOTA *Los programadores de C/C++ notarán que Java hace mucho más sencillo el traducir entre el nombre legible de una constante enumerada y su valor binario. Ésta es una ventaja significante de la implementación de enumeraciones en Java.*

Las enumeraciones en Java son tipos de clase

Como se explicó una enumeración de Java es un tipo de clase. Aunque no se instancia un **enum** utilizando **new**, éstos tienen casi las mismas capacidades de las clases. El hecho de que **enum** defina una clase hace que la enumeración de Java tenga poderes que en una enumeración en otros lenguajes simplemente no existen. Por ejemplo, se pueden tener constructores, agregar variables de instancia y métodos, e incluso implementar interfaces.

Es importante entender que cada constante de la enumeración es un objeto de su propio tipo enumerado. Así, cuando se define un constructor para un **enum**, el constructor es llamado cuando cada constante de enumeración es creada. También, cada constante de enumeración tiene su propia copia de cualquier variable de instancia definida para la enumeración. Por ejemplo, consideremos la siguiente versión de la enumeración **Manzana**:

```
//Uso de constructores, variables y métodos en una enumeración.
enum Manzana {
  Jonathan(10), GoldenDel(9), RedDel(12), Winesap(15), Cortland(8);

  private int price; // precio de cada Manzana

  // constructor
  Manzana (int p) { price = p; }

  int getPrice () { return price; }
}

class EnumDemo3
  public static void main (String args[])
  {
  Manzana ap;
```

```
    // mostrar el precio de Winesap
    System.out.println("Winesap cuesta " +
                       Manzana.Winesap.getPrice() +
                       " centavos.\n");

    // mostrar todos los tipos de manzana y su precio.
    System.out.println( "Todas las manzanas y sus precios: ");
    for(Manzana a : Manzana.values())
      System.out.println(a+ " cuesta " + a.getPrice() +
                         " centavos.");
  }
}
```

La salida de este programa se muestra a continuación:

```
Winesap cuesta 15 centavos

Todas las manzanas y sus precios:
Jonathan cuesta 10 centavos
GoldenDel cuesta 9 centavos
RedDel cuesta 12 centavos
Winesap cuesta 15 centavos
Cortland cuesta 8 centavos
```

Esta versión de **Manzana** agrega tres cosas. La primera es la variable de instancia **precio**, la cual es utilizada para almacenar el precio de cada tipo de manzana. La segunda es el constructor **Manzana**, al cual se pasa el precio de cada manzana. La tercera es el método **getPrice( )**, el cual regresa el valor del **precio**.

Cuando se declara la variable **ap** en **main( )**, el constructor **Manzana** es llamado una vez para cada constante especificada. Nótese como los argumentos para el constructor son especificados dentro de paréntesis al lado de cada constante, como se muestra a continuación:

```
Jonathan (10), GoldenDel (9), RedDel (12), Winesap (15), Cortland (8);
```

Estos valores son pasados al parámetro **p** de **Manzana( )**, el cual asigna el valor a la variable **precio**. El constructor es llamado una vez para cada constante.

Dado que cada constante en la enumeración tiene su propia copia de la variable **precio**, es posible obtener el precio de un tipo específico de manzana llamando al método **getPrice( )**. Por ejemplo, en el método **main( )** el precio de **Winesap** es obtenido por la siguiente llamada:

```
Manzana.Winesap.getPrice()
```

El precio para cada una de la variedades es obtenido utilizando un ciclo a través de la enumeración con un ciclo **for**. Debido a que hay una copia de **precio** para cada constante en la enumeración, el valor asociado con una constante es independiente del valor asociado con otra. Éste es un concepto poderoso, y sólo está disponible cuando las enumeraciones son implementadas como clases tal como lo hace Java.

Aunque el ejemplo anterior contiene sólo un constructor, una **enumeración** puede tener dos o más constructores sobrecargados, tal como las otras clases lo pueden hacer. Por ejemplo, la siguiente versión de **Manzana** provee un constructor por omisión que inicializa el precio a –1, para indicar que no existe un precio disponible:

```
// Uso de constructores en enumeraciones
enum Manzana {
  Jonathan(10), GoldenDel (9), RedDel, Winesap (15), Cortland (8) ;

  private int price; // precio de cada manzana

  // constructor
  Manzana (int p) { price = p; }

  // constructor sobrecargado
  Manzana () { price = -1; }

  int getPrice () { return price; }
}
```

Nótese que en esta versión, para **RedDel** no se proporcionan argumentos. Esto significa que el constructor por omisión es llamado, y la variable precio para **RedDel** tendrá el valor –1.

Existen dos restricciones que se aplican a las **enumeraciones**. Primero, una enumeración no puede heredar de otra clase. En segundo lugar, una enumeración no puede ser una superclase. Esto significa que una enumeración no puede ser extendida. Por lo demás, una enumeración actúa de igual forma que cualquier otro tipo de clase. La clave es recordar que cada constante en la enumeración es un objeto de la clase en la cual está definida.

Las enumeraciones heredan de la clase enum

Aunque no se puede heredar a una superclase cuando se declara un **enum**, todas las enumeraciones automáticamente heredan una: **java.lang.Enum**. Esta clase define varios métodos que están disponibles para el uso de todas las enumeraciones. La clase **Enum** se describe a detalle en la Parte II, por ahora revisaremos sólo tres de sus métodos.

Es posible obtener la posición de una constante en la enumeración, también llamado su *valor ordinal*, llamando al método **ordinal( )**, definido como:

final int ordinal( )

Este método regresa el valor ordinal de la constante que lo invoca. Los valores ordinales comienzan en cero. Así, en la enumeración **Manzana**, **Jonathan** tiene un valor ordinal cero, **GoldenDel** tiene un valor ordinal 1, **RedDel** tiene un valor ordinal 2, y así sucesivamente.

Es posible comparar los valores ordinales de dos constantes de la misma enumeración utilizando el método **compareTo( )**. El cual está definido como:

final int compareTo(*tipoEnum e*)

Donde *tipoEnum* es el tipo de la enumeración, y *e* es la constante a comparar con la constante que la invoca al método. Recuerde que la constante que invoca y la constante *e* deben ser del mismo tipo de enumeración. Si la constante que invoca tiene un valor ordinal menor que *e*, entonces **compareTo( )** regresa un valor negativo. Si los dos valores ordinales son iguales, entonces se regresa cero. Si la constante que invoca tiene un valor mayor que *e*, entonces se regresa un valor positivo.

Es posible comparar la igualdad de una constante de enumeración con cualquier otro objeto utilizando **equals( )**, este método sobrescribe al método **equals( )** definido por la clase **Object**. Aunque **equals( )** puede comparar una constante de enumeración con cualquier otro objeto, esos dos objetos serán iguales sólo si ambos hacen referencia a la misma constante, dentro de

la misma enumeración. El simple hecho de tener valores ordinales en común no causará que **equals( )** regrese el valor de verdad si las dos constantes son de diferentes enumeraciones.

Es posible comparar dos referencias enumeración en busca de igualdad utilizando **==**. El siguiente programa muestra el uso de los métodos **ordinal( )**, **compareTo( )** y **equals( )**:

```
// Ejemplo de los métodos ordinal(), compareTo(), y equals().

// Una enumeración de variedades de manzana
enum Manzana {
  Jonathan, GoldenDel, RedDel, Winesap, Cortland
}

class EnumDemo4 {
  public static void main(String args[])
  {
    Manzana ap, ap2, ap3;

    // Obtener todos los valores ordinales utilizando el método ordinal().
    System.out.println("Estas son todas las constantes manzana" +
                       "y sus valores ordinales: ");

    for(Manzana a : Manzana.values())
      System.out.println(a+ " " + a.ordinal());

    ap = Apple.RedDel;
    ap2 = Apple.GoldenDel;
    ap3 = Apple.RedDel;

    // uso de los métodos compareTo() y equals()
    if(ap.compareTo(ap2) < 0)
      System.out.println(ap + " va antes de " + ap2);

    if(ap.compareTo(ap2) > 0)
      System.out.println(ap2 + " va antes de " + ap);

    if(ap.compareTo(ap3) == 0)
      System.out.println(ap + " es igual a " + ap3);

    System.out.println() ;

    if(ap.equals(ap2))
      System.out.println("¡Error!") ;

    if(ap.equals(ap3))
      System.out.println(ap + " es igual a " + ap3);

    if (ap == ap3)
      System.out.println(ap + " == " + ap3);

  }
}
```

La salida del programa se muestra a continuación:

```
Estas son todas las constantes manzana y sus valores ordinales:
Jonathan 0
GoldenDel 1
```

```
RedDel 2
Winesap3
Cortland 4

GoldenDel va antes de RedDel
RedDel es igual a RedDel

RedDel es igual a RedDel
RedDel == RedDel
```

Otro ejemplo con enumeraciones

Antes de continuar, veamos un ejemplo diferente que utiliza **enum**. En el Capítulo 9 se construyó un programa de toma de decisiones automáticas. En esa versión, las variables llamadas **NO**, **SI**, **QUIZAS**, **DESPUES**, **PRONTO**Y **NUNCA**, fueron declaradas dentro de una interfaz y utilizadas para representar las posibles respuestas. Técnicamente no hay ningún error con esa solución; sin embargo, la enumeración es una mejor opción. A continuación se muestra una versión mejorada de ese programa, la cual utiliza una **enumeración** llamada **Respuestas** para definir las posibles respuestas. Se recomienda al lector comparar esta versión con la original del Capítulo 9.

```
// Una versión mejorada del programa de "Toma de Decisiones"
// escrito en el capítulo 9. Esta versión utiliza una
// enumeración, en vez de variables de interfaz para
// representar los valores de las respuestas.

import java.util.Random;

// Una enumeración de posibles respuestas.
enum Respuestas {
  NO, SI, QUIZAS, DESPUES, PRONTO, NUNCA
}

class Question {
  Random rand = new Random();
  Respuestas ask ( ) {
    int prob = (int) (100 * rand.nextDouble());

    if (prob < 15)
      return Respuestas.QUIZAS; // 15%
    else if (prob < 30)
      return Respuestas.NO; // 15%
    else if (prob < 60)
      return Respuestas.SI; // 30%
    else if (prob < 75)
      return Respuestas.DESPUES; // 15%
    else if (prob < 98)
      return Respuestas.PRONTO; / / 13%
    else
      return Respuestas.NUNCA; // 2%
  }
}
```

```
class AskMe {
  static void answer(Respuestas result) {
    switch (result) {
      case NO:
        System.out.println("No") ;
        break;
      case SI:
        System.out.println("Si") ;
        break;
      case QUIZAS:
        System.out.println("Quizás") ;
        break;
      case DESPUES:
        System.out.println("Después") ;
        break;
      case PRONTO:
        System.out.println("Pronto");
        break;
      case NUNCA:
        System.out.println("Nunca") ;
        break;
    }
  }

  public static void main(String args[]) {
    Question q = new Question() ;
    answer(q.ask()) ;
    answer(q.ask()) ;
    answer(q.ask()) ;
    answer(q.ask()) ;
  }
}
```

Envoltura de tipos

Como sabemos, Java utiliza tipos primitivos (también llamados tipos simples), tales como **int** y **double**, como los tipos de datos básicos del lenguaje. Los tipos primitivos son utilizados para favorecer el rendimiento. Utilizar objetos para valores primitivos agregaría una sobrecarga, incluso para cálculos simples, poco deseable. Por ello, los tipos primitivos no son parte de la jerarquía de objetos y por ende no heredan de la clase **Object**.

A pesar de los beneficios de rendimiento ofrecidos por los tipos primitivos, existen ocasiones cuando se requiere su representación como un objeto. Por ejemplo, no es posible pasar como parámetro a un método un tipo primitivo por referencia. Además, muchas de las estructuras de datos estándares implementadas por Java trabajan sobre objetos, lo que significa que no es posible usar estas estructuras de datos para almacenar datos primitivos. Para gestionar estas situaciones (y otras) Java provee la envoltura de tipos, que consiste en proporcionar clases que encapsulan a un *tipo primitivo* dentro de un objeto. Las clases que sirven como envolturas de tipos son descritas a detalle en la Parte II, pero son introducidas aquí debido a que están relacionadas directamente con la característica de autoboxing de Java.

Las envolturas de tipos son **Double**, **Float**, **Long**, **Integer**, **Short**, **Byte**, **Character** y **Boolean**. Estas clases ofrecen un conjunto amplio de métodos que permiten integrar completamente a los tipos primitivos dentro de la jerarquía de objetos de Java. Cada uno es examinado brevemente a continuación.

Character

Character es la envoltura del tipo **char**. El constructor para **Character** es

```
Character (char ch)
```

Donde *ch* especifica el carácter que será envuelto por el objeto **Character** que está siendo creado. Para obtener el valor **char** contenido en el objeto **Character**, se llama al método **charValue( )**, como se muestra a continuación:

```
char charValue( )
```

el cuál regresa al carácter encapsulado.

Boolean

Boolean es la envoltura alrededor de los valores del tipo primitivo **boolean**. El cual define estos constructores:

```
Boolean (boolean boolValue)
Boolean (String boolString)
```

En la primera versión, *boolValue* debe ser **true** o **false**. En la segunda versión, si *boolString* contiene la cadena "true" (en minúsculas o mayúsculas), entonces el nuevo objeto **Boolean** será verdadero, de otra forma, será falso.

Para obtener el valor del objeto **Boolean**, se utiliza el método **booleanValue( )**, como se muestra a continuación:

```
boolean booleanValue( )
```

el cual regresa el valor de tipo **boolean** equivalente al del objeto invocado.

Las envolturas de tipos numéricos

Por mucho, las envolturas más comúnmente usadas son aquellas que representan valores numéricos. Estas envolturas son **Byte**, **Short**, **Integer**, **Long**, **Float** y **Double**. Todas las envolturas de los tipos numéricos heredan de la clase abstracta **Number**. **Number** declara métodos que regresan el valor de un objeto en cada uno de los diferentes formatos. Estos métodos se muestran a continuación:

```
byte byteValue( )
double doubleValue( )
float floatValue( )
int intValue( )
long longValue( )
short shortValue( )
```

Por ejemplo, **doubleValue( )** regresa el valor de un objeto como un valor de tipo **double**, **floatValue( )** regresa el valor como un valor de tipo **float**, y así sucesivamente. Estos métodos son implementados por cada una de las envolturas de tipos numéricos.

Todas las envolturas de tipos numéricos definen constructores que permiten a un objeto ser construido a partir de un valor dado o a partir de una cadena que represente el valor. Por ejemplo, aquí se presentan los constructores definidos para la clase **Integer**:

Integer(int *num*)
Integer(String *str*)

Si *str* no contiene un valor numérico válido entonces una excepción de tipo **NumberFormatException** es lanzada. Todas las envolturas de tipo sobrescriben al método **toString( )**. El cual regresa en una forma compresible el valor contenido dentro de la envoltura. Esto permite, por ejemplo, desplegar el valor del objeto envuelto cuando es usado en un **println( )** sin tener que convertirlo a su tipo primitivo.

El siguiente programa demuestra cómo se utiliza una envoltura de tipo numérico para encapsular un valor y después extraerlo.

```
// demostración de envoltura de tipos
class Wrap {
  public static void main(String args[]) {

    Integer iOb = new Integer(100);

    int i = iOb.intValue();

    System.out.println(i + " " + iOb); // muestra 100 100
  }
}
```

Este programa envuelve el valor entero de 100 dentro de un objeto **Integer** llamado **iOb**. El programa entonces obtiene ese valor llamando **intValue( )** y almacena el resultado en **i**.

El proceso de encapsulación de un valor dentro de un objeto es llamado *boxing*. Así, en el programa, esta línea realiza el boxing del valor 100 dentro de un objeto **Integer**.

```
Integer iOb = new Integer(100);
```

El proceso de extracción del valor desde una envoltura de tipos es llamado *unboxing*. Por ejemplo, el programa realiza unboxing del valor de **iOb** con la siguiente línea:

```
int i = iOb.intValue();
```

El mismo procedimiento general utilizado por el programa anterior para boxing y unboxing ha sido empleado desde la versión original de Java. Sin embargo, con la llegada del JDK 5, Java mejoró considerablemente esta característica adicionando el concepto de autoboxing que se describe a continuación.

Autoboxing

A partir de JDK 5, Java agregó dos importantes características: *autoboxing* y *auto-unboxing*. Autoboxing es el proceso por medio del cual un tipo primitivo es automáticamente encapsulado dentro de un objeto generado por envoltura de tipos, en cualquier lugar donde un objeto de ese tipo se requiera. No es necesario construir explícitamente un objeto. Auto-unboxing es el proceso mediante el cual el valor de un objeto (generado por envoltura de tipos) es automáticamente despojado de su envoltura de tipo cuando el valor es requerido. No es necesario llamar a un método tal como **intValue( )** o **doubleValue( )**.

Agregar autoboxing y auto-unboxing estiliza enormemente el código de muchos algoritmos, removiendo el tedio del boxing y unboxing manual de valores. Esto también ayuda a prevenir errores. Además, es muy importante para la implementación de tipos parametrizados, la cual opera sólo en objetos. Finalmente, autoboxing hace el trabajo con el Framework de Colecciones (descrita en la Parte II) mucho más sencilla.

Con el autoboxing ya no se necesita construir manualmente un objeto para envolver un tipo primitivo. Sólo se necesita asignar el valor a una referencia de una envoltura del tipo. Java automáticamente construye el objeto. Por ejemplo, ésta es la forma moderna de construir un objeto **Integer** que envuelve al valor de 100:

```
Integer iOb = 100; // autoboxing de un valor de tipo int
```

Note que no se crea explícitamente un objeto usando la palabra clave **new**. Java gestiona esto automáticamente.

Para realizar el unboxing de un objeto, simplemente debe asignar el objeto referenciado a una variable de tipo primitivo. Por ejemplo, para realizar unboxing de **iOb**, se utiliza la siguiente línea:

```
int i = iOb; //auto-unboxing
```

Java gestiona los detalles automáticamente.

Ésta es una nueva versión del programa anterior re-escrito utilizando autoboxing / unboxing:

```
// Ejemplo de autoboxing / unboxing
class AutoBox {
  public static void main (String args []) {

    Integer iOb = 100; / / autoboxing un valor de tipo int

    int i = iOb; // auto-unboxing

    System.out.println(i+ " " + iOb); // muestra 100 100
  }
}
```

Autoboxing y métodos

Además de ocurrir en los casos simples de asignación de valores, el autoboxing ocurre en cualquier momento que un tipo primitivo debe ser convertido en un objeto y auto-unboxing toma lugar cuando un objeto debe ser convertido a un tipo primitivo. Así, autoboxing y auto-unboxing pueden ocurrir cuando un argumento se pasa a un método, o cuando un valor es devuelto por un método. Por ejemplo, considere el siguiente código:

```
// autoboxing y auto-unboxing ocurren cuando
// un método recibe argumentos o devuelve valores

class AutoBox2 {
  // este método recibe un argumento del tipo Integer y regresa
  // un valor del tipo primitivo int
  static int m (Integer v) {
    return v ; // auto-unboxing el objeto v a un valor int
  }
```

```
  pub1ic static void main(String args[]) {
    //  Se envía un valor de tipo int al método m()
        y asigna el valor a un objeto Integer.
    // El argumento 100 sufre autoboxing,
    // al igual que el valor regresado por el método
    Integer iOb = m(100);

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

El programa despliega el siguiente resultado:

```
100
```

En el programa, note que **m( )** especifica un parámetro **Integer** y regresa un valor de tipo **int** como resultado. Dentro del **main( )**, al método **m( )** se le pasa el valor 100. Dado que **m( )** está esperando un **Integer**, al valor 100 se aplica autoboxing. Luego el método **m( )** regresa el valor **int** equivalente a su argumento. Esto causa que la variable **v** sufra auto-unboxing. Finalmente, este valor **int** es asignado al objeto **iOb** en **main( )**, el cuál causa que el valor **int** regresado pase nuevamente por autoboxing.

Autoboxing en expresiones

En general, autoboxing y auto-unboxing ocurren en cualquier momento en que una conversión de un valor a un objeto o de un objeto a un valor es requerida. Esto aplica también a las expresiones. Dentro de una expresión a los objetos se les aplica automáticamente unboxing y al resultado de la expresión autoboxing si es necesario. Por ejemplo, consideremos el siguiente programa:

```
// autoboxing y auto-unboxing ocurren en las expresiones.

class AutoBox3 {
  pub1ic static void main(String args[]) {

    Integer iOb, iOb2;
    int i;

    iOb = 100;
    System.out.println("Valor original de iOb: " + iOb);

    // El código siguiente aplica automáticamente unboxing a iOb,
    // realiza un incremento y luego aplica autoboxing nuevamente
    // para colocar el resultado en iOb
    ++iOb;
    System.out.println("Después de ++iOb: " + iOb);

    // La expresión se evalúa después de que a iOb se le aplica unboxing,
    // al resultado se le aplica autoboxing y luego se almacena en iOb2.
    iOb2 = iOb + (iOb / 3);
    System.out.println("iOb2 después de evaluar la expresión es: " + iOb2);

    // La misma expresión se evalúa ahora sin que sea necesario
```

```
    // aplicar autoboxing al resultado
    i = iOb + (iOb / 3);
    System.out.println ("i después de evaluar la expresión es: " + i);

  }
}
```

La salida se muestra a continuación:

```
    Valor original de iOb: 100
    Después de ++iOb: " + 101
    iOb2 después de evaluar la expresión es: 134
    i después de evaluar la expresión es: 134
```

En el programa es importante poner especial atención en la siguiente línea:

```
++iOb;
```

Esta línea causa que el valor en **iOb** sea incrementado. Funciona de la siguiente forma: **iOb** pasa por el proceso de unboxing, el valor es incrementado, y al resultado se le aplica autoboxing.

El proceso de auto-unboxing también permite que se mezclen diferentes tipos de objetos numéricos en una expresión. Una vez que los valores pasan por unboxing, se aplican las conversiones y promociones estándares. Por ejemplo, el siguiente programa es perfectamente válido:

```
c1ass AutoBox4 {
  pub1ic static void main(String args[])

    // autoboxing y auto-unboxing dentro de expresiones

    Integer iOb = 100;
    Doub1e dOb = 98.6;

    dOb = dOb + iOb;
    System.out.println("dOb después de la expresión: " + dOb);
  }
}
```

La salida de ese código es:

```
    dOb después de la expresión: 198.6
```

Como se puede ver, tanto el objeto **Double dOb** como el objeto Integer **iOb** participan en la adición y el resultado pasa por autoboxing antes de ser almacenado en **dOb**.

Debido al auto-unboxing, es posible utilizar objetos numéricos enteros para controlar una sentencia **switch**. Por ejemplo, considere el siguiente segmento de código:

```
Integer iOb = 2;

switch (iOb) (
  case 1: System.out.println ("uno") ;
    break;
  case 2: System.out.println ("dos") ;
    break;
```

```
    default: System.out.println("error");
}
```

Cuando la expresión en el **switch** es evaluada, a **iOb** se le aplica unboxing y su valor **entero** es obtenido.

Los ejemplos muestran cómo la aplicación de autoboxing y auto-unboxing a objetos numéricos dentro de expresiones es intuitiva y fácil. En el pasado, un código similar habría involucrado conversión de tipos y llamadas a métodos, como por ejemplo **intValue( )**.

Autoboxing en valores booleanos y caracteres

Como se describió anteriormente, Java también proporciona envolturas para los tipos primitivos **boolean** y **char**. Esas envolturas son **Boolean** y **Character**. Los procesos de autoboxing y auto-unboxing se aplican a esas envolturas también. Por ejemplo, considere el siguiente programa:

```
// Autoboxing y unboxing de objetos Boolean y Character.

class AutoBox5 {
  public static void main(String args[]) {

    // autoboxing y unboxing aplicado a un valor boolean.
    Boolean b = true;

    // b pasa por auto-unboxing cuando es utilizada
    // en una expresión condicional
    if(b) System.out.println("b es true");

    // autoboxing y unboxing aplicado a un valor char.
    Character ch = 'x'; // autoboxing un char
    char ch2 = ch; // unboxing un char

    System.out.println("ch2 es " + ch2);
  }
}
```

La salida se muestra a continuación:

```
b es true
ch2 es x
```

El punto más importante a considerar en este programa es el auto-unboxing de **b** dentro de la expresión condicional **if**. Como recordará, las expresiones condicionales que controlan un **if** deben ser evaluadas a un resultado de tipo **boolean**. Debido al auto-unboxing, el valor **boolean** que está contenido en **b** es obtenido automáticamente cuando la expresión condicional lo requiere. La llegada del autoboxing y el unboxing ha permitido que un objeto **Boolean** pueda ser utilizado para controlar una sentencia **if**.

Debido al auto-unboxing, un objeto de tipo **Boolean** ahora también puede ser utilizado para controlar cualquiera de las sentencias de ciclo de Java. Cuando un objeto **Boolean** es utilizado en la expresión condicional de un **while**, **for** o **do/while**, se le aplica automáticamente unboxing para convertirlo en su equivalente **boolean**. Por ejemplo el siguiente código es perfectamente válido:

```
Boolean b;
// ...
while (b) { // . . .
```

Autoboxing y la prevención de errores

Además de las facilidades que ofrecen, también ayudan a prevenir errores. Por ejemplo, consideremos el siguiente programa:

```
// Aquí se produce un error debido al unboxing manual
class UnboxingError {
  public static void main(String args []) {

    Integer iOb = 1000; // autoboxing del valor 1000

    int i = iOb.byteValue(); // ¡unboxing manual como tipo byte !

    System.out.println(i); // ¡esto NO desplegará 1000!
  }
}
```

El programa no despliega el valor esperado 1000, en su lugar despliega –24. La razón es que el valor dentro de la variable **iOb** pasa por un unboxing manualmente por la llamada al método **byteValue( )**, el cual causa el truncamiento del valor 1000 almacenado en **iOb**. Esto da como resultado que el valor –24 sea asignado a **i**. El auto-unboxing previene este tipo de errores porque el valor en **iOb**, mediante auto-unboxing, dará lugar a un valor compatible con **int**.

Comúnmente, autoboxing siempre crea el objeto correcto, y auto-unboxing siempre produce el valor correcto, no hay forma de que el proceso produzca un tipo de objeto o un valor incorrecto. En los casos excepcionales donde se requiere un tipo diferente del que es arrojado por el proceso automático, es posible realizar manualmente boxing y unboxing de los valores. Claro que, los beneficios del autoboxing y unboxing se perderían. En general, los nuevos programas deberían utilizar autoboxing y unboxing. Es la forma en que los programas modernos de Java serán escritos.

Una advertencia sobre el uso autoboxing

Ahora que Java incluye autoboxing y auto-unboxing, podría resultar tentador utilizar objetos tales como **Integer** o **Double** y abandonar a los tipos primitivos del todo. Por ejemplo, con autoboxing y unboxing es posible escribir código como éste:

```
// uso incorrecto de autoboxing y unboxing
Double a, b, c;

a = 10.0;
b = 4.0;

c = Math.sqrt(a*a + b*b);

System.out.println("La hipotenusa es: " + c) ;
```

En este ejemplo, objetos de tipo **Double** contienen los valores que son usados para calcular la hipotenusa del triángulo rectángulo. Aunque este código es técnicamente correcto y de hecho funciona correctamente, hace un muy mal uso del autoboxing y unboxing. Es mucho menos eficiente que el código equivalente escrito utilizando el tipo primitivo **double**. La razón es que cada aplicación de autoboxing y auto-unboxing agrega trabajo adicional que no se presenta cuando se usan tipos primitivos.

En general, el uso de la envoltura de tipos debe restringirse solamente a los casos en los cuales la representación de un objeto de un tipo primitivo sea requerida. Autoboxing y unboxing no fueron agregados a Java para eliminar los tipos primitivos.

Anotaciones (metadatos)

A partir de JDK 5, una nueva característica fue agregada a Java la cual permite incrustar información suplementaria dentro de un archivo fuente. Esta información, llamada *anotación*, no cambia las acciones del programa. Una anotación no cambia la semántica de un programa. Sin embargo, la información de la anotación puede ser usada por varias herramientas durante las etapas de desarrollo e implementación. Por ejemplo, una anotación puede ser procesada por un generador de código fuente. El término *metadato* también es utilizado para referirse a esta característica, pero el término *anotación* es más descriptivo y se utiliza más comúnmente.

Fundamentos de las anotaciones

Una anotación se crea a través de un mecanismo basado en una **interfaz**. Comencemos con un ejemplo. Aquí está la declaración para una anotación llamada **MiAnotacion**:

```
// un tipo simple de anotación
@interface MiAnotacion {
  String str();
  int val();
}
```

Primero, observe que la palabra clave **interface** está precedida por una @. Esto le dice al compilador que estamos declarando un tipo de anotación. Ahora, observe los dos miembros **str( )** y **val( )**. Todas las anotaciones consisten únicamente en declaraciones de métodos para los cuales no se provee cuerpo alguno. Java implementa esos métodos. Además, los métodos actúan más como campos, como se verá a continuación.

Una anotación no puede incluir una cláusula **extends**. Sin embargo, todos los tipos de anotación automáticamente extienden a la interfaz **Annotation**. La interfaz **Annotation** es una super interfaz de todas las anotaciones y está declarada dentro del paquete **java.lang.annotation**. La interfaz sobrescribe los métodos **hashCode( )**, **equals( )** y **toString( )** definidos por la clase **Object**, además define al método **annotationType( )** el cual regresa un objeto de tipo **Class** que representa a la anotación que hizo la invocación.

Una vez que se ha declarado es posible utilizar la anotación. Cualquier tipo de declaración puede tener una anotación asociada. Por ejemplo, clases, métodos, campos, parámetros y constantes **enumeradas** pueden tener anotaciones asociadas. Incluso una anotación puede tener anotaciones asociadas. En todos los casos, la anotación precede al resto de la declaración.

Cuando se aplica una anotación, se dan valores a sus miembros. Por ejemplo, a continuación un ejemplo de la anotación **MiAnotacion** aplicada a un método:

```
// Aplicando una anotación a un método
@MiAnotacion (str = "Ejemplo de Anotación", val = 100)
public static void miMetodo() { // …
```

Esta anotación se liga al método **miMetodo( )**. Observe cuidadosamente la sintaxis de la anotación. El nombre de la anotación está precedido por una @ y seguido por una lista de inicialización de miembros entre paréntesis. Para darle un valor a un miembro, al nombre del

miembro se les asigna un valor. Además, en el ejemplo, la cadena "Ejemplo de Anotación" se asigna al miembro **str** de **MiAnotacion**. Nótese que no hay paréntesis después de **str** en esta asignación. Cuando a un miembro de la anotación se le da un valor, sólo se escribe su nombre. Así que los miembros de la anotación parecen campos en este contexto.

Especificación de la política de retención

Antes de explorar más a fondo a las anotaciones, es necesario discutir la *política de retención de las anotaciones*. Una política de retención determina en qué punto una anotación es desechada. Java define tres políticas al respecto, las cuales se encuentran encapsuladas en la enumeración java**.lang.annotation.RetentionPolicy.** Dichas políticas son **SOURCE**, **CLASS** y **RUNTIME**.

Una anotación con una política de retención **SOURCE** es conservada sólo en el archivo fuente y es descartada durante la compilación.

Una anotación con una política de retención **CLASS** es almacenada en el archivo **.class** durante la compilación. Sin embargo, no está disponible a través de la máquina virtual de Java durante el tiempo de ejecución.

Una anotación con una política de retención de **RUNTIME** es almacenada en el archivo **.class** durante la compilación y está disponible a través de la máquina virtual de Java durante el tiempo de ejecución. Así, la retención **RUNTIME** ofrece la persistencia más grande para una anotación.

Una política de retención para una anotación se especifica utilizando una de las anotaciones predefinidas de Java: **@Retention**. Su forma general se muestra a continuación:

@Retention(*política*)

Aquí, *política* debe ser una de las constantes enumeradas discutidas previamente. Si no se especifica una política de retención para una anotación, la política utilizada por omisión es **CLASS**.

La siguiente versión de **MiAnotacion** utiliza **@Retention** para especificar la política de retención **RUNTIME**. Así, **MiAnotacion** estará disponible para la máquina virtual de Java durante la ejecución del programa.

```
@Retention (RetentionPolicy.RUNTIME)
@interface MiAnotacion{
  String str();
  int val();
}
```

Obtención de anotaciones en tiempo de ejecución

Aunque las anotaciones están diseñadas en su mayor parte para ser utilizadas por otras herramientas de desarrollo e implementación, si se especifica una política de retención de **RUNTIME**, entonces pueden ser requeridas en tiempo de ejecución por cualquier programa de Java utilizando *reflexión*. La reflexión es la característica que permite que información acerca de la clase sea obtenida en tiempo de ejecución. El API de reflexión está contenido en el paquete **java.lang.reflect**. Existe un gran número de formas de utilizar reflexión y no todas se examinarán aquí. Sin embargo, veremos algunos ejemplos que aplican a las anotaciones.

El primer paso para usar reflexión es obtener un objeto del tipo **Class** que represente la clase a la cual pertenecen las anotaciones que se desean obtener. **Class** es una de las clases

predefinidas en Java, está definida en **java.lang**, y se describe a detalle en a Parte II de este libro. Existen varias formas de obtener un objeto de tipo **Class**, una de las más fáciles es llamando al método **getClass( )**, el cuál está definido en la clase **Object**. Su forma general es:

final Class getClass( )

Esta línea regresa el objeto de tipo **Class** que representa al objeto invocado. **getClass( )** y muchos otros métodos relativos a la reflexión hacen uso de las características de tipos parametrizados. Sin embargo, dado que la característica de tipos parametrizados será discutida hasta el Capítulo 14, estos métodos son mostrados y usados aquí en su forma más cruda. Como resultado, se nos estará presentando un mensaje de advertencia cuando compilemos los programas siguientes. En el Capítulo 14, aprenderemos sobre los tipos parametrizados.

Después de obtener un objeto de tipo **Class**, podemos utilizar sus métodos para obtener información sobre los elementos declarados por la clase, incluyendo sus anotaciones. Si se desea obtener las anotaciones asociadas con un elemento específico declarado dentro de una clase, se debe en primer lugar obtener un objeto que representa dicho objeto. Por ejemplo, **Class** provee (entre otros) los métodos **getMethod( )**, **getField( )** y **getConstructor( )**, los cuales obtienen información acerca de un método, campo y constructor respectivamente. Estos métodos regresan objetos de tipo **Method**, **Field** y **Constructor**.

Para entender el proceso, trabajemos con un ejemplo que obtiene las anotaciones asociadas con un método. Para hacer eso, primero se debe obtener un objeto **Class** que representa la clase y entonces llamar a **getMethod( )** en ese objeto **Class**, especificando el nombre del método. **getMethod( )** tiene esta forma general:

Method getMethod(String *nombreMetodo*, Class … *parametroTipos*)

El nombre del método se pasa a través de *nombreMetodo*. Si el método tiene argumentos, entonces será necesario especificar objetos de tipo **Class** que representen esos tipos utilizando *parametroTipos*. Observe que *parametroTipos* es un parámetro varargs. Esto significa que se pueden especificar tantos tipos de parámetros como sea necesario, incluyendo cero. **getMethod( )** regresa un objeto **Method** que representa el método. Si el método no está presente se lanza una excepción del tipo **NoSuchMethodException**.

Para los objetos **Class**, **Method**, **Field** o **Constructor**, se pueden obtener sus anotaciones asociadas llamando al método **getAnnotation( ).** Su forma general se muestra a continuación:

Annotation getAnnotation(Class *tipoAnotacion*)

Donde *tipoAnotacion* es un objeto de tipo **Class** que representa a la anotación en la cual estamos interesados. El método regresa una referencia a la anotación. Utilizando esta referencia, se pueden obtener los valores asociados con los miembros de la anotación. El método regresa **null** si la anotación no es encontrada, lo cual ocurriría si la anotación no tiene una retención **RUNTIME**.

A continuación se presenta un programa que ensambla todas las piezas mostradas anteriormente y utiliza reflexión para mostrar la anotación asociada con un método.

```
import java.lang.annotation.*;
import java.lang.reflect.*;

// Declaración de un tipo de anotación
@Retention(RetentionPolicy.RUNTIME)
@interface MiAnotacion {
```

```
  String str();
  int val () ;
}

class Meta {

  // colocar una anotación a un método
  @MiAnotacion(str = "Anotación de Ejemplo", val = 100)
  public static void miMetodo() {
    Meta ob = new Meta();

    // Obtener la anotación del método
    // y desplegar los valores de sus miembros.
    try {

    // Primero, se obtiene un objeto de tipo Class que representa
    // a la clase
    Class c = ob.getClass ();

    // Ahora, se obtiene un objeto de tipo Method que representa
    // a este método
    Method m = c.getMethod ("miMetodo") ;

    // Luego, se obtiene la anotación
    MiAnotacion a = m.getAnnotation(MiAnotacion.class);

    // Finalmente, se muestran los valores
    System.out.println(a.str() + " " + a.val ());
  } catch (NoSuchMethodException exc) {
    System.out.println ("método no encontrado.");
  }
}

  public static void main (String args []) {
    miMetodo () ;
  }
}
```

La salida del programa se muestra a continuación:

```
    Anotación de Ejemplo 100
```

Este programa utiliza reflexión, como se describió, para obtener y desplegar los valores de **str** y **val** de la anotación **MiAnotacion** asociada con **miMetodo( )** in la clase **Meta**. Debemos poner atención en dos aspectos particulares. Primero, en la línea:

```
MiAnotacion a = m.getAnnotation(MiAnotacion.class);
```

observe la expresión **MiAnotacion.class**. Esta expresión proporciona un objeto de tipo **Class** para la anotación de tipo **MiAnotacion**. Esta construcción se denomina *literal de clase*. Es posible utilizar este tipo de expresión en cualquier momento que un objeto **Class** para una clase conocida sea necesario. Por ejemplo, esta sentencia pudo ser utilizada para obtener el objeto **Class** para **Meta**:

```
Class c = Meta.class;
```

Claro está que esto sólo funciona cuando se conoce el nombre de la clase de un objeto de manera anticipada, lo cual podría no siempre ser el caso. En general, es posible obtener una literal de clase para clases, interfaces, tipos primitivos y arreglos.

El segundo punto de interés es la forma en que los valores asociados con **str** y **val** son obtenidos para ser mostrados por la siguiente línea:

```
System.out.println(a.str () + " " + a.val() );
```

Note que son invocados utilizando la sintaxis de llamada a métodos. Esta misma forma se utiliza para obtener el valor de cualquier miembro de una anotación.

Un segundo ejemplo de reflexión

En el ejemplo anterior, **miMetodo( )** no tiene parámetros. Así que cuando se llamó a **getMethod( )** sólo se pasó como parámetro el nombre del método. Sin embargo, para obtener un método que tiene parámetros se deben especificar objetos de tipo **Class** representando los tipos de esos parámetros como argumentos para **getMethod( )**. Como ejemplo veamos una versión ligeramente diferente del programa anterior:

```
import java.lang.annotation.*;
import java.lang.reflect.*;

@Retention(RetentionPolicy.RUNTIME)
@interface MiAnotacion {
  String str();
  int val () ;
}

class Meta {

  // miMetodo ahora tiene dos argumentos
  @MiAnotacion(str = "Dos parámetros", val = 19)
  public static void miMetodo(String str, int i)
  {
    Meta ob = new Meta() ;

    try {
      Class c = ob.getClass();

      // Aquí se especifican los tipos de los parámetros
      Method m = c.getMethod("miMetodo",String.class, int.class);

      MiAnotacion a = m.getAnnotation(MiAnotacion.class);

      System.out.println(a.str()+" "+ a.val());
    } catch (NoSuchMethodException exc) {
      System.out.println("método no encontrado.");
    }
  }

  public static void main(String args[]) {
    miMetodo("prueba", 10);
  }
}
```

La salida de esta versión se muestra a continuación:

```
Dos parámetros 19
```

En esta versión **miMetodo( )** toma un parámetro **String** y un parámetro **int**. Para obtener información de este método, **getMethod( )** debe ser llamado como se muestra a continuación:

```
Method m = c.getMethod("miMetodo", String.class, int.class);
```

donde los objetos **Class** para **String** e **int** son enviados como argumentos adicionales.

Obteniendo todas las anotaciones

Se pueden obtener todas las anotaciones que tienen retención **RUNTIME** y que está asociadas a algún elemento, llamando al método **getAnnotations( )** sobre ese elemento. **getAnnotations( )** tiene la siguiente forma general:

```
Annotation[ ] getAnnotations( )
```

Esto regresa un arreglo con las anotaciones. **getAnnotations( )** puede ser llamado por objetos de tipo **Class**, **Method**, **Constructor** y **Field**.

Aquí está otro ejemplo de reflexión que muestra como obtener todas las anotaciones asociadas con una clase y con un método. Se declaran dos anotaciones y luego se utilizan esas anotaciones en una clase y en un método.

```
// Mostrar todas las anotaciones de una clase y un método
import java.lang.annotation.*;
import java.lang.reflect.*;

@Retention(RetentionPolicy.RUNTIME)
@interface MiAnotacion {
  String str();
  int val();
}

@Retention(RetentionPolicy.RUNTIME)
@interface What {
  String description();
}

@What(description = "Una prueba de anotación para clase")
@MiAnotacion(str = "Meta2", val = 99)
class Meta2 {

  @What(description = "Una prueba de anotación en método")
  @MiAnotacion(str = "Probando", val = 100)
  public static void miMetodo() {
    Meta2 ob = new Meta2();

    try {
      Annotation annos[] = ob.getClass() .getAnnotations();

      // Mostrar todas las anotaciones para Meta2.
      System.out.println("Todas las anotaciones para Meta2:");
      for(Annotation a : annos)
        System.out.println(a);

      System.out.println();

      // Mostrar todas las anotaciones para miMetodo.
      Method m = ob.getClass( ).getMethod("miMetodo");
      annos = m.getAnnotations();

      System.out.println("Todas las anotaciones para miMetodo:");
      for(Annotation a : annos)
```

```
        System.out.println(a) ;
    } catch (NoSuchMethodException exc) {
      System.out.println("método no encontrado");
    }
  }
  public static void main(String args[]) {
    miMetodo () ;
  }
}
```

La salida del programa anterior es:

```
Todas las anotaciones para Meta2:
@What(description = "Una prueba de anotación para clase")
@MiAnotacion(str = "Meta2", val = 99)

Todas las anotaciones para miMetodo:
@What(description = "Una prueba de anotación en método")
@MiAnotacion(str = "Probando", val = 100)
```

El programa utiliza **getAnnotations( )** para obtener un arreglo con todas las anotaciones asociadas con la clase **Meta2** y con el método **miMetodo( )**. Como se explicó, **getAnnotations( )** regresa un arreglo de objetos **Annotation**. Recuerde que **Annotation** es una super-interfaz de todas las interfaces de anotaciones y que sobrescribe al método **toString( )** de la clase **Object**. Así, cuando se imprime en pantalla una referencia a una **Annotation**, se llama al método **toString( )** para generar una cadena que describe a la anotación, como se muestra en la salida del ejemplo anterior.

La interfaz AnnotatedElement

Los métodos **getAnnotation( )** y **getAnnotations( )** utilizados en los ejemplos anteriores se definen en la interfaz **AnnotatedElement**, la cual está definida en **java.lang.reflect**. Esta interfaz proporciona reflexión para anotaciones y es implementada por las clases **Method**, **Field**, **Constructor**, **Class** y **Package**.

Además de **getAnnotation( )** y **getAnnotations( )**, **AnnotatedElement** define otros dos métodos. El primero es **getDeclaredAnnotations( )**, que tiene la siguiente forma general:

```
Annotation[ ] getDeclaredAnnotations( )
```

Este método regresa todas las anotaciones no heredadas presentes en el objeto que realiza la invocación. El segundo método es **isAnnotationPresent( )**, el cual tiene la siguiente forma general:

```
boolean isAnnotationPresent (Class tipoAnotacion)
```

Éste devuelve verdadero si la anotación especificada por *tipoAnotacion* está asociada con el objeto que realiza la invocación, en caso contrario devuelve falso.

NOTA *Los métodos **getAnnotation( )** y **isAnnotationPresent( )** hacen uso de la característica de tipos parametrizados para garantizar la seguridad de tipos. Dado que los tipos parametrizados serán revisados hasta el capítulo 14, sus firmas se muestran en este capítulo en su forma más cruda.*

Utilizando valores por omisión

Se pueden dar valores por omisión a los miembros de las anotaciones para que sean utilizados si no se especifica un valor cuando la anotación es aplicada. Un valor por omisión se especifica agregando una cláusula **default** a la declaración de un miembro. La forma general de la declaración es:

tipo miembro( ) default *valor;*

Donde, *valor* debe ser de un tipo compatible con el *tipo* del miembro.

Esta versión de la anotación **@MiAnotacion** incluye valores por omisión:

```
// Declaración de un tipo de anotación que incluye valores por omisión
@Retention(RetentionPolicy.RUNTIME)
@interface MiAnotacion {
  String str () default "Probando";
  int val() default 9000;
}
```

Esta declaración da un valor por omisión de"Probando"a **str** y 9000 a **val**. Esto significa que ningún valor necesita ser especificado cuando se utiliza **@MiAnotacion.** Sin embargo a ambos se les puede dar valores si se desea. Éstas son las cuatro formas en que **@MiAnotacion** puede ser usada:

```
@MyAnno() // str y val toman valores por omisión
@MyAnno(str = "algún texto") // val toma el valor por omisión
@MyAnno(val = 100) // str toma el valor por omisión
@MyAnno(str = "Probando" , val = 100) // ningún miembro toma valores por
omisión.
```

El siguiente programa ejemplifica el uso de los valores por omisión en una anotación.

```
import java.lang.annotation.*;
import java.lang.reflect.*;

// Una declaración de tipo de anotación con valores por omisión en sus miembros
@Retention(RetentionPolicy.RUNTIME)
@interface MiAnotacion {
  String str () default "Probando";
  int val() default 9000;
}

class Meta3 {

  // Aplicando una anotación con valores por omisión a un método
  @MiAnotacion ()
  public static void miMetodo() {
    Meta3 ob = new Meta3();

    // Obtener las anotaciones asociadas al método
    // y desplegar los valores de sus miembros
    try {
      Class c = ob.getClass();

      Method m = c.getMethod("miMetodo");
```

```
      MiAnotacion a = m.getAnnotation(MiAnotacion.class);

      System.out.println (a.str () + " " + a.val ());
    } catch (NoSuchMethodException exc) {
       System.out.println("método no encontrado");
    }
  }

  public static void main(String args[]) {
    miMetodo() ;
  }
}
```

La salida del código anterior es;

```
Probando 9000
```

Anotaciones de marcado

Las anotaciones de *marcado* son un tipo especial de anotaciones que no contienen miembros. Su único propósito es marcar una declaración. Es decir, su presencia como anotación es suficiente. La mejor forma de determinar si una anotación de marcado está presente es utilizando el método **isAnnotationPresent( )**, el cual está definido por la interfaz **AnnotatedElement**.

A continuación un ejemplo que usa una anotación de marcado. Debido a que una interfaz de marcado no contiene miembros, el sólo determinar si está presente o no es suficiente.

```
import java.lang.annotation.*;
import java.lang.reflect.*;

// Una anotación de marcado
@Retention(RetentionPolicy.RUNTIME)
@interface MyMarker { }

class Marker {

  // Aplicamos la anotación anterior sobre un método
  // Observe que los paréntesis no son necesarios
  @MyMarker
  public static void miMetodo() {
    Marker ob = new Marker() ;

    try {
      Method m = ob. getClass ().getMethod ("miMetodo") ;

      // Se determina si la anotación está presente
      if(m.isAnnotationPresent(MyMarker.class))
        System.out.println("La anotación de marcado está presente");

    } catch (NoSuchMethodException exc) {
      System.out.println("método no encontrado");
    }
  }
```

```
  public static void main(String args[]) {
    miMetodo() ;
  }
}
```

La salida de este programa, mostrada a continuación, confirma que **@MyMaker** está presente:

```
La anotación de marcado está presente
```

En el programa, observe que no se necesita colocar paréntesis al lado de **@MyMaker** al aplicarlo. Así, **@MyMaker** es aplicado simplemente anotando su nombre, como sigue:

```
@MyMaker
```

No está mal suministrar un par de paréntesis vacíos, pero no son necesarios.

Anotaciones de un solo miembro

Las anotaciones de *un solo miembro* contienen solamente un miembro y funcionan como una anotación normal excepto por el hecho de que permiten una forma corta de especificar el valor del miembro. Cuando solamente un miembro está presente, es posible especificar simplemente el valor para dicho miembro cuando la anotación es aplicada y no es necesario especificar el nombre del miembro. Sin embargo, para el uso de esta forma corta, el nombre del miembro debe ser la palabra **value** tal cual.

El siguiente ejemplo crea y usa una anotación de un solo miembro:

```
import java.lang.annotation.*;
import java.lang.reflect.*;

// Una anotación de un solo miembro
@Retention(RetentionPolicy.RUNTIME)
@interface MySingle {
  int value(); // el nombre de esta variable debe ser value
}

class Single {

  // Se aplica la anotación anterior sobre un método.
  @MySingle(l00)
  public static void miMetodo() {
    Single ob = new Single();

    try {
      Method m = ob.getClass().getMethod("miMetodo");

      MySingle anno = m.getAnnotation(MySingle.class);

      System.out.println(anno.value()); // mostrará 100

    } catch (NoSuchMethodException exc) {
       System.out.println("método no encontrado");
    }
  }
```

```
  public static void main(String args[]) {
    miMetodo();
  }
}
```

Como era de esperarse, este programa despliega el valor de 100. En el programa, **@MySingle** se utiliza en **miMetodo( )**, como se muestra a continuación:

```
@MySingle(100)
```

Observe que el **signo** = no necesita ser especificado.

Es posible utilizar la misma sintaxis para anotaciones que tiene más miembros, pero todos los miembros adicionales deben tener valores por omisión. Por ejemplo, aquí agregamos al miembro **xyz** con un valor por omisión de cero.

```
@interface UnaAnotacion{
  int value();
  int xyz() default 0;
}
```

En los casos donde se desea usar el valor por omisión de **xyz**, se puede aplicar **@unaAnotacion**, como se muestra a continuación, simplemente especificando el valor del miembro **value** utilizando la sintaxis de anotación con un solo miembro.

```
@UnaAnotacion(88)
```

En este caso, **xyz** tiene el valor por omisión de cero, y **value** tiene el valor 88. Claro está, que especificar un valor diferente para **xyz** requiere que ambos miembros sean explícitamente nombrados, como se muestra a continuación

```
@UnaAnotacion(value = 88, xyz = 99)
```

Recuerde, en cualquier momento que se esté utilizando la anotación de un solo miembro, el nombre de dicho miembro debe ser value.

Anotaciones predefinidas en Java

Java define varias anotaciones predefinidas. La mayoría son especializadas, pero siete son de propósito general. De esas siete, cuatro son importadas de **java.lang.annotation: @Retention**, **@Documented**, **@Target**, y **@Inherited**. Tres, **@Override**, **@Deprecated** y **@SupressWarnings** están incluidas en **java.lang**. Cada una se describe a continuación.

@Retention

@Retention está diseñada para ser usada sólo como una anotación a otra anotación. Ésta especifica la política de retención como se describió anteriormente en este capítulo.

@Documented

La anotación **@Documented** es una interfaz de marcado que le dice a una herramienta que una anotación sea documentada. Está diseñada para ser usada solo como una anotación para una declaración de anotación.

@Target

La anotación **@Target** especifica el tipo de declaraciones en las cuales una anotación puede ser aplicada. Está diseñada para ser utilizada únicamente como una anotación a otra anotación.

@Target toma un argumento, el cual debe ser una constante de la enumeración **ElementType**. Este argumento especifica el tipo de declaración en el cual la anotación puede ser aplicada. Las constantes se muestran junto con el tipo de declaración al cual corresponden.

Constante	La anotación puede ser aplicada a
ANNOTATION_TYPE	Otra anotación
CONSTRUCTOR	Constructor
FIELD	Campo
LOCAL_VARIABLE	Variable local
METHOD	Método
PACKAGE	Paquete
PARAMETER	Parámetro
TYPE	Clase, interfaz o enumeración

Se puede especificar uno o más de estos valores en una anotación **@Target**. Para especificar múltiples valores, se deben especificar en una lista delimitada con llaves. Por ejemplo, para especificar que una anotación aplica sólo a campos y variables locales, se puede usar la siguiente anotación:

```
@Target ( {ElementType.FIELD, ElementType.LOCAL_VARIABLE})
```

@Inherited

@Inherited es una anotación de marcado que puede ser usada solo sobre declaraciones de anotaciones y afecta únicamente a anotaciones que serán utilizadas en la declaración de clases. **@Inherited** causa que la anotación de una super clase sea heredada por sus subclases. Por consiguiente, cuando una solicitud por una anotación específica es hecha a la subclase, si la anotación no está presente en la subclase entonces se revisará en la superclase. Si en la superclase está presente la anotación y además tiene especificada la anotación **@Inherited**, entonces la anotación se devolverá como resultado de la solicitud.

@Override

@Override es una anotación de marcado y puede ser utilizada sólo en métodos. Un método anotado con **@Override** debe sobrescribir un método de una superclase. Si no existe, se tendrá como resultado un error en tiempo de compilación. Esto se utiliza para garantizar que un método de una superclase es sobrescrito, y no solo sobrecargado.

@Deprecated

@Deprecated es una anotación de marcado que indica que una declaración es obsoleta y ha sido reemplazada por una nueva forma.

@SupressWarnings

@SupressWarnings especifica que una o más advertencias que podrían ser generadas por el compilador serán suprimidas. Las advertencias a suprimir son especificadas por su nombre en una cadena. Esta anotación puede ser aplicada a cualquier tipo de declaración.

Restricciones para las anotaciones

Existen algunas restricciones que se aplican a la declaración de anotaciones. Primero, ninguna anotación puede heredar de otra. Segundo, todos los métodos declarados por una anotación deben ser métodos sin parámetros. Además, deben devolver lo siguiente:

- Un tipo primitivo, como un **int** o un **double**.
- Un objeto de tipo **String** o **Class**.
- Un tipo **enum**.
- Otro tipo de anotación.
- Un arreglo de elementos de uno de los tipo mencionados anteriormente.

Las anotaciones no pueden utilizar tipos parametrizados. Los tipos parametrizados se describen en el Capítulo 14. Finalmente, los métodos de una anotación no pueden especificar la cláusula **throws**.

CAPÍTULO 13

E/S, applets y otros temas

Este capítulo introduce dos de los paquetes más importantes de Java: **io** y **applet**. El paquete **io** contiene el sistema básico de E/S (entradas/salidas) de Java, incluyendo la E/S con archivos. El paquete **applet** gestiona los applets. La gestión de las E/S y de los applets se realiza mediante bibliotecas del API de Java, y no mediante palabras reservadas del lenguaje. Por este motivo, en la Parte II de este texto, que examina la interfaz de las clases de Java, se analizan en profundidad estos dos tópicos. Este capítulo examina las bases de estos dos subsistemas, de forma que se ve cómo se integran en el lenguaje Java, tanto en la programación como en su entorno de ejecución. Este capítulo también examina las últimas palabras claves de Java: **transient**, **volatile**, **instanceof**, **native**, **strictfp** y **assert**. Y concluye examinando la combinación de palabras clave **static import** y un uso adicional de la palabra clave **this**.

Fundamentos de E/S

En los programas ejemplo que aparecen en los anteriores doce capítulos no se ha hecho mucho uso del subsistema de E/S. De hecho, en dichos ejemplos, aparte de los métodos **print( )** y **println( )**, no se ha usado de manera significativa ningún otro de los métodos de E/S. La razón es simple y es que en la mayor parte de las aplicaciones reales de Java no se utilizan programas cuya salida sea basada en texto por consola, sino que son aplicaciones gráficas que basan su interacción con el usuario en un conjunto de herramientas gráficas denominado AWT (por sus siglas en inglés, Abstract Window Toolkit) o Swing. Aunque los programas con salida basada en texto son ideales en la enseñanza del lenguaje, no se utilizan en programas reales. Además, la E/S por consola es bastante limitada y engorrosa incluso en programas sencillos. Por todo ello, la E/S basada en texto por consola no es muy importante en la programación en Java.

A pesar de lo expuesto en el párrafo anterior, Java proporciona un sistema de E/S completo y flexible en lo referente a archivos y redes. El sistema de E/S de Java es coherente y consistente. De hecho, una vez que se entienden sus fundamentos, el resto del sistema de E/S se domina fácilmente.

Flujos

Los programas en Java realizan las E/S a través de flujos. Un *flujo* es una abstracción de una entidad que produce o consume información. Un flujo está ligado a un dispositivo físico por el sistema de E/S de Java. Todos los flujos se comportan de igual manera, incluso en el caso de que los dispositivos

físicos reales a los que están ligados sean diferentes. Por lo tanto, las mismas clases y métodos de E/S se pueden aplicar a cualquier tipo de dispositivo. Esto significa que un flujo de entrada se puede utilizar para distintos tipos de entrada: un archivo de disco, el teclado o una conexión de red. Del mismo modo, un flujo de salida se puede referir a la consola, a un archivo de disco o a una conexión de red. Los flujos son una forma clara y sencilla de tratar las entradas/salidas sin que el código tenga que tener en cuenta, por ejemplo, si el dispositivo es un teclado o la red. Java implementa los flujos en una jerarquía de clases definida en el paquete **java.io.**

Flujos de bytes y flujos de caracteres

Java define dos tipos de flujos: de bytes y de caracteres. Los *flujos de bytes* proporcionan un medio conveniente para gestionar la entrada y salida de bytes. Los flujos de bytes se utilizan, por ejemplo, cuando se escriben o leen datos binarios. Los *flujos de caracteres*, por el contrario, son adecuados para gestionar la entrada y salida de caracteres. Utilizan el código Unicode y, por lo tanto, se pueden utilizar internacionalmente. En algunos casos, los flujos de caracteres pueden ser más eficientes que los flujos de bytes.

La versión inicial de Java (Java 1.0) no incluía el flujo de caracteres; esto implica que todas la E/S estaban orientadas a byte. El flujo de caracteres fue añadido por Java 1.1.Y esto hizo que se desecharan algunas clases y métodos orientados a byte. Por este motivo, en algunos casos, resulta apropiado actualizar códigos antiguos que no utilizaban el flujo de caracteres, para aprovechar la ventaja que éste tiene.

Conviene también tener en cuenta que, en el más bajo nivel, todas las E/S están orientadas bytes. El flujo basado en caracteres simplemente proporciona un medio conveniente y eficaz para el manejo de estos.

En los siguientes apartados se presenta una visión general de los flujos orientados a byte y de los flujos orientados a carácter.

Las clases de flujos de bytes

Los flujos de bytes se definen mediante dos jerarquías de clases. En el nivel superior hay dos clases abstractas: **InputStream** y **OutputStream**. Cada una de estas clases abstractas tiene varias subclases no abstractas que gestionan las diferencias entre los diversos dispositivos tales como, archivos de disco, conexiones de red, e incluso espacios de memoria. Las clases referentes a flujos de bytes se muestran en la Tabla 13-1. Sólo unas pocas de estas clases se discuten más adelante, en este apartado. Las demás se describen en la Parte II. Recuerde que para utilizar las clases de flujos se debe importar el paquete **java.io**.

Las clases abstractas **InputStream** y **OutputStream** definen varios métodos que las otras clases implementan. Dos de los más importantes son los métodos **read( )** y **write( )**, que permiten, respectivamente, leer y escribir bytes de datos. Ambos métodos se declaran como abstractos dentro de **InputStream** y **OutputStream** y son sobrescritos en las clases derivadas.

Las clases de flujos de caracteres

Los flujos de caracteres se definen mediante dos jerarquías de clases. En el nivel más alto se encuentran las clases abstractas, **Reader** y **Writer**. Estas clases gestionan el flujo de caracteres Unicode. Java define varias subclases de estas dos clases. Las clases referentes a flujos de caracteres se muestran en la Tabla 13-2.

Las clases abstractas **Reader** y **Writer** definen varios métodos que las otras clases implementan. Dos de los métodos más importantes son los métodos **read( )** y **write( )**, que

sirven para leer y escribir caracteres, respectivamente. Estos métodos son sobrescritos en las clases derivadas.

Clase	Significado
BufferedInputStream	Flujo de entrada con buffer
BufferedOutputStream	Flujo de salida con buffer
ByteArrayInputStream	Flujo de entrada que lee desde un arreglo de bytes
ByteArrayOutputStream	Flujo de salida que escribe en un arreglo de bytes
DataInputStream	Flujo de entrada que tiene métodos para leer los tipos primitivos o básicos de Java
DataOutputStream	Flujo de salida que tiene métodos para escribir los tipos primitivos o básicos de Java
FileInputStream	Flujo de entrada que lee desde un archivo
FileOutputStream	Flujo de salida que escribe en un archivo
FilterInputStream	Implementa **InputStream**
FilterOutputStream	Implementa **OutputStream**
InputStream	Clase abstracta que define un flujo de entrada
ObjectInputStream	InputStream para objetos
ObjectOutputStream	OutputStream para objetos
OutputStream	Clase abstracta que define un flujo de salida
PipeInputStream	Canal de entrada
PipeOutputStream	Canal de salida
PrintStream	Flujo de salida que contiene los métodos **print()** y **println()**
PushbackInputStream	Flujo de entrada que permite -cuando se ha leído un byte- que se devuelva de nuevo al flujo de entrada
RandomAccessFile	Permite el acceso aleatorio a un archivo de E/S
SequenceInputStream	Flujo de entrada que es una combinación de dos o más flujos de entrada que serán leídos secuencialmente, uno después de otro

TABLA 13-1 Clases de flujos de bytes

Clase	Significado
BufferedReader	Flujo de entrada de caracteres con buffer
BufferedWriter	Flujo de salida de caracteres con buffer
CharArrayReader	Flujo de entrada que lee desde un arreglo de caracteres
CharArrayWriter	Flujo de salida que escribe en un arreglo de caracteres
FileReader	Flujo de entrada que lee desde un archivo
FileWriter	Flujo de salida que escribe en un archivo
FilterReader	Filtro de lectura
FilterWriter	Filtro de escritura

TABLA 13-2 Clases de flujos de caracteres

Clase	Significado
InputStreamReader	Flujo de entrada que convierte bytes a caracteres
LineNumberReader	Flujo de entrada que cuenta las líneas
OutputStreamWriter	Flujo de salida que convierte caracteres a bytes
PipedReader	Canal de entrada
PipedWriter	Canal de salida
PrintWriter	Flujo de salida que contiene los métodos **print()** y **println()**
PushbackReader	Flujo de entrada que permite regresar caracteres a un flujo de entrada
Reader	Clase abstracta que define un flujo de entrada de caracteres
StringReader	Flujo de entrada que lee desde un String
StringWriter	Flujo de salida que escribe en un String
Writer	Clase abstracta que define un flujo de salida de caracteres

TABLA 13-2 Clases de flujos de caracteres (*continuación*)

Flujos predefinidos

Como sabemos, todos los programas de Java importan automáticamente el paquete **java.lang.** Este paquete define una clase denominada **System**, que encapsula algunos aspectos del entorno de ejecución. Por ejemplo, utilizando algunos de sus métodos, se puede obtener la hora actual o los valores de diversas propiedades asociadas al sistema. **System** también contiene tres variables con flujos predefinidos: **in**, **out** y **err**. Estos campos se declaran como **public**, **static** y **final** en la clase **System**. Esto significa que pueden ser utilizadas por cualquier parte del programa sin necesidad de una referencia a un objeto específico de tipo **System**.

System.out hace referencia al flujo de salida estándar, que, por omisión, es la consola. **System.in** se refiere al flujo de entrada estándar, que, por omisión, es el teclado. **System.err** se refiere al flujo de error estándar, que, también por defecto, es la consola. Sin embargo, cualquiera de estos flujos puede ser redirigido a cualquier dispositivo compatible de E/S.

System.in es un objeto del tipo **InputStream**; **System.out** y **System.err** son objetos del tipo **PrintStream**. Todos estos son flujos de bytes, aunque se utilizan normalmente para leer y escribir caracteres desde y en la consola. Como se verá, estos flujos se pueden envolver en flujos basados en caracteres, si se desea.

En los capítulos anteriores ya se ha utilizado **System.out** en los ejemplos. Se puede utilizar **System.err** de la misma forma, pero, como se explica en el próximo apartado, el uso de **System.in** es un poco más complicado.

Entrada por consola

En Java 1.0, la única forma de realizar la entrada por consola era mediante un flujo de bytes, y un código que utilice este enfoque sigue siendo válido. Hoy en día, utilizar un flujo de bytes para leer una entrada por consola es todavía técnicamente posible, pero este procedimiento

no es recomendable. En Java 2, se aconseja utilizar un flujo basado en caracteres para leer una entrada por consola, ya que de esta forma resulta más sencillo internacionalizar y mantener el código.

En Java, la entrada por consola se lleva a cabo leyendo de **System.in**. Para obtener un flujo basado en caracteres conectado a la consola, se envuelve **System.in** en un objeto **BufferedReader**. **BufferedReader** proporciona un buffer para el flujo de entrada. El constructor que se utiliza comúnmente es:

BufferedReader(Reader *entrada*)

Donde *entrada* es el flujo que será ligado a la instancia de **BufferedReader** que se está siendo creada. **Reader** es una clase abstracta. Una de sus subclases concretas es **InputStreamReader**, que convierte bytes en caracteres. Para obtener un objeto **InputStreamReader** ligado a **System.in**, se utiliza el siguiente constructor:

InputStreamReader (InputStream *entrada*)

Como **System.in** se refiere a un objeto del tipo **InputStream**, se puede utilizar en el *lugar de entrada*. La siguiente línea de código realiza las dos acciones anteriores para crear un objeto **BufferedReader** conectado al teclado:

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

Después de la ejecución de esta sentencia, **br** es un flujo basado en caracteres ligado a la consola a través de **System.in.**

Lectura de caracteres

Para leer un carácter desde un **BufferedReader**, se utiliza el método **read( ).** La versión de **read( )** que utilizaremos es:

int read( ) throws IOException

Cada vez que se llame a **read( )**, este método lee un carácter del flujo de entrada y lo devuelve como un valor entero. Cuando encuentra el final del flujo, devuelve el valor -1. También puede lanzar una excepción del tipo **IOException**.

El siguiente programa muestra un ejemplo en el que se utiliza el método **read( )** para leer caracteres de la consola hasta que el usuario pulsa la letra "q". Observe que cualquier excepción de E/S que se genere es lanzada fuera de **main( )**. Este manejo es común cuando se lee información de la consola, aunque la excepción puede ser gestionada, si se desea.

```
// Uso de un BufferedReader para leer caracteres de la consola.
import java.io.*;

class BRRead {
  public static void main(String args[])
    throws IOException
  {
    char c;
    BufferedReader br = new
```

```
            BufferedReader(new InputStrearnReader(System.in));
    System.out.println("Introduzca caracteres, pulse 'q' para salir.");

    // lectura de caracteres
    do{
      c = (char) br.read();
      System.out.println(c);
    } while (c!= 'q');
  }
}
```

Como ejemplo de ejecución de este programa, se presenta la siguiente salida:

```
Introduzca caracteres, pulse 'q' para salir.
123abcq
1
2
3
a
b
c
q
```

Esta salida puede ser ligeramente distinta de la esperada. Esto es así porque, por omisión, **System.in** es un flujo con buffer. Esto significa que realmente no se pasa ninguna entrada al programa hasta que no se pulsa la tecla ENTER. Como se puede imaginar, esto hace que **read( )** no sea de mucha utilidad para la entrada interactiva por consola.

Lectura de cadenas

Para leer una cadena desde el teclado, se usa la versión del método **readLine( )** que es miembro de la clase **BufferedReader**. Su forma general es la siguiente:

String readLine( ) throws IOException

Como se puede ver, este método devuelve un objeto **String**.

El siguiente programa es un ejemplo del uso de la clase **BufferedReader** y del método **readLine( )**; el programa lee e imprime líneas de texto hasta que se escribe la palabra "stop":

```
// Lectura de una cadena desde la consola utilizando la clase BufferedReader.
import java.io.*;

class BRReadLines {
  public static void main(String args[])
    throws IOException
  {
    // Se crea un objeto BufferedReader usando System.in
    BufferedReader br = new BufferedReader(new
                            InputStreamReader(System.in));
    String str;

    System.out.println("Introduzca las líneas de texto.");
    System.out.println("Introduzca 'stop' para salir.");
    do{
```

```
      str = br.readLine();

      System.out.println(str);
    } while(!str.equals("stop"));
  }
}
```

El siguiente ejemplo crea un pequeño editor de texto utilizando un arreglo de objetos **String,** y después lee líneas de texto, almacenándolas en el arreglo. Leerá hasta un máximo de 100 líneas o hasta que se introduzca la palabra "stop". En el ejemplo se utiliza un objeto de la clase **BufferedReader** para leer de la consola.

```
// Un pequeño editor.
import java.io.*;

class PequeñoEditor {
  public static void main(String args[])
    throws IOException
  {
    // se crea un BufferedReader usando System.in
    BufferedReader br = new BufferedReader(new
                          InputStreamReader(System.in));
    String str[] = new String[l00];

    System.out.println("Introduzca las líneas de texto.");
    System.out.println("Introduzca 'stop' para salir.");
    for(int i=0; i<l00; i++) {
      str[i] = br.readLine();
      if(str[i] .equals("stop")) break;
    }

    System.out.println("\nEste es su archivo:");

    // presenta las líneas
    for(int i=0; i<l00; i++) {
      if(str[i].equals("stop")) break;
      System.out.println(str[i]);
    }
  }
}
```

Un ejemplo de la ejecución de este programa es el siguiente:

```
Introduzca las líneas de texto.
Introduzca 'stop' para salir.
Esta es la primera línea.
Esta es la segunda línea.
Java facilita el trabajo con cadenas.
Simplemente cree objetos de tipo String.
stop
Este es su archivo:
Esta es la primera línea.
Esta es la segunda línea.
Java facilita el trabajo con cadenas.
Simplemente cree objetos de tipo String.
```

Salida por consola

La salida por consola se realiza fácilmente con los métodos **print( )** y **println( )**, descritos anteriormente y que son los más utilizados en los ejemplos de este libro. Estos métodos están definidos en la clase **PrintStream** (que es el tipo de objeto al que hace referencia **System.out**). Aunque **System.out** es un flujo de bytes, su uso es viable para las salidas sencillas de un programa. En el siguiente apartado, sin embargo, se describe una alternativa basada en caracteres.

Como **PrintStream** es un flujo de salida derivado de **OutputStream**, también implementa el método de bajo nivel **write( )**. Por tanto, el método **write( )** se puede utilizar para escribir en la consola. La forma más sencilla de **write( )** definida por **PrintStream** es la siguiente:

void write(int *valorByte*)

Este método escribe el byte especificado por *valorByte*. Aunque este valor se declara como un entero, sólo se escriben los ocho bits de orden más bajo. El siguiente programa utiliza el método **write( )** para presentar en la pantalla el carácter "A" seguido de una nueva línea:

```
// Ejemplo de System.out.write().
class WriteDemo {
  public static void main(String args[]) {
    int b;

    b = 'A';
    System.out.write(b);
    System.out.write(' \n') ;
  }
}
```

Normalmente no se usa el método **write( )** para la salida por consola (aunque puede ser útil en algunas situaciones) debido los métodos **print( )** y **println( )** son más fáciles de usar.

La clase PrintWriter

Aunque el uso de **System.out** está permitido para escribir en la consola, su uso se recomienda para depurar programas o para programas de ejemplo, como los que aparecen en este libro. Para programas reales, el método recomendado para escribir en la consola consiste en utilizar un flujo de tipo **PrintWriter**. **PrintWriter** es una de las clases basada en caracteres de Java. Utilizarla hace que resulte más fácil la internacionalización del programa.

PrintWriter define varios constructores. El que se utiliza aquí es el siguiente:

PrintWriter(*OutputStream* flujosalida, boolean *flushOnNuevaLinea*)

Aquí, *flujosalida* es un objeto del tipo **OutputStream**, y *flushOnNuevalinea*, una variable boolean que controla si Java limpia el flujo de salida cada vez que se llama al método **println( )**. Si la variable *flushOnNuevaLinea* es **true**, el flujo de salida se limpia automáticamente. En caso de que sea **false**, esta limpieza no es automática.

La clase **PrintWriter** soporta los métodos **print( )** y **println( )** para todos los tipos incluyendo **Object**. Por tanto, se pueden utilizar estos métodos de la misma forma que se han utilizado con

System.out. Si un argumento no es de un tipo simple, los métodos de **PrintWriter** llaman al método **toString( )** del objeto, y entonces se imprime el resultado.

Para escribir en la consola utilizando la clase **PrintWriter**, hay que especificar **System.out** como flujo de salida y limpiar el flujo después de cada nueva línea. Por ejemplo, esta línea de código crea un objeto **PrintWriter** conectado a la salida por consola:

```
PrintWriter pw = new PrintWriter(System.out, true);
```

La siguiente aplicación muestra cómo se utiliza un objeto **PrintWriter** para manejar la salida por consola:

```
// Ejemplo de PrintWriter
import java.io.*;

public class printWriterDemo {
  public static void main(String args[]) {
    PrintWriter pw = new PrintWriter (System.out, true);
    pw.println("Esto es una cadena");
    int i = -7;
    pw.println(i) ;
    double d = 4.5e-7;
    pw.println(d);
  }
}
```

La salida de este programa es la siguiente:

```
Esto es una cadena
-7
4.5E-7
```

Recuerde que se puede utilizar **System.out** para escribir un texto sencillo en la consola cuando se está aprendiendo Java o depurando programas. Sin embargo, un **PrintWriter** hará que las aplicaciones reales se puedan internacionalizar más fácilmente. No obstante, en el texto se seguirá utilizando **System.out** para escribir en la consola, ya que en los programas de ejemplo que se muestran en este libro no representa ninguna ventaja utilizar un **PrintWriter**.

Lectura y escritura de archivos

Java proporciona clases y métodos que permiten leer y escribir archivos. En Java, todos los archivos están orientados a bytes, y Java proporciona métodos que permiten leer y escribir datos en un archivo. Sin embargo, Java también permite envolver un flujo de archivo orientado a bytes en un objeto basado en caracteres. Esta técnica se describe en la Parte II. Este capítulo sólo examina las cuestiones básicas de la E/S de archivos.

Dos de las clases relacionadas con flujos que más se utilizan son **FileInputStream** y **FileOutputStream**, que crean flujos de bytes asociados a archivos. Para abrir un archivo, simplemente hay que crear un objeto de una de estas dos clases, especificando el nombre del archivo como argumento del constructor. Aunque ambas clases proporcionan diferentes constructores sobrecargados, la forma más utilizada en este texto es la siguiente:

FileInputStream(String *nombreArchivo*) throws FileNotFoundException
FileOutputStream(String *nombreArchivo*) throws FileNotFoundException

nombreArchivo es el nombre del archivo que se trata de abrir. Si al crear un flujo de entrada, el archivo indicado no existe, entonces se lanza una excepción del tipo **FileNotFoundException**. En el caso del flujo de salida, si no se puede crear el archivo se genera una excepción del tipo **FileNotFoundException**. Cuando se abre un archivo de salida se destruye cualquier archivo que existiera antes con ese mismo nombre.

Cuando se ha terminado de trabajar con un archivo, hay que cerrarlo llamando al método **close( )**, que está definido en las clases **FileInputStream** y **FileOutputStream**, así:

```
void close( ) throws IOException
```

Para leer de un archivo, se puede utilizar una versión del método **read( )** definida en **FileInputStream**. La que utilizaremos aquí es la siguiente:

```
int read( ) throws IOException
```

Cada vez que se llama a este método, se lee un solo byte del archivo y se devuelve un valor entero. El método **read( )** devuelve el valor –1 cuando se encuentra el final archivo. También puede lanzar una excepción del tipo **IOException**.

El siguiente programa utiliza el método **read( )** para leer y mostrar el contenido de un archivo de texto cuyo nombre se pasa como argumento en la línea de comandos. Observe los bloques **try/catch** que gestionan los dos errores que podrían producirse al utilizar este programa (que no se encontrara el archivo especificado o que el usuario olvidase incluir el nombre del archivo). Este mismo planteamiento se puede utilizar siempre que se utilicen argumentos en la línea de comandos. Otras excepciones de E/S que podrían ocurrir se lanzan fuera de **main( )**, lo cual es correcto en este sencillo ejemplo. Sin embargo en la vida real el programador deberá gestionar todas las excepciones de E/S en sus programas.

```
/*  Muestra un archivo de texto.

    Para usar este programa, hay que especificar
    el nombre del archivo que se desee ver.
    Por ejemplo, para ver un archivo llamado TEST.TXT,
    utilice la siguiente línea de comandos.

    java ShowFile TEST.TXT

*/

import java.io.*;

class ShowFile {
  public static void main(String args[])
    throws IOException
  {
    int i;
    FileInputStream fin;

    try {
      fin = new FileInputStream(args[0]);
    } catch (FileNotFoundException e) {
      System.out.println("Archivo no encontrado");
      return;
```

```
    } catch{ArraylndexOutOfBoundsException e) {
      System.out.println("Para utilizar el programa escriba: ShowFile Archivo");
      return;
    }

    // Lee caracteres hasta que se encuentra el fin del archivo
    do {
      i = fin.read();
      if(i != -1) System.out.print((char) i);
    } while(i != -1);
    fin.close ();
  }
}
```

Para escribir en un archivo se usará el método **write( )**, definido por **FileOutputStream**. Su forma más simple es la siguiente:

void write(int *valorByte*) throws IOException

Este método escribe el byte especificado por *valorByte* en el archivo. Aunque *valorByte* se declara como un entero, sólo se escriben en el archivo los ocho bits de orden más bajo. Si se produce un error durante la escritura, se lanza una excepción del tipo **IOException**. El siguiente ejemplo utiliza el método **write( )** para copiar un archivo de texto:

```
/*  Copia de un archivo de texto.

    Para usar este programa, se debe especificar el nombre
    del archivo fuente y del archivo destino.
    Por ejemplo, para copiar un archivo llamado PRIMER.TXT
    en un archivo llamado SEGUNDO.TXT, use la siguiente
    línea de comandos.

    java CopyFile PRIMER.TXT SEGUNDO.TXT
*/

import java.io.*;

class CopyFile {
  public static void main(String args[])
    throws IOException
  {
    int i;
    FilelnputStream fin;
    FileOutputStream fout;

    try {
      // abre el archivo de entrada
      try {
        fin = new FilelnputStream(args[0]);
        } catch(FileNotFoundException e) {
          System.out.println("Archivo de entrada no encontrado");
          return;
        }
```

```
      // Se abre el archivo de salida
      try {
        fout = new FileOutputStream(args[1]);
      } catch(FileNotFoundException e) {
        System.out.println("Error al abrir el archivo de salida");
        return;
      }
    } catch(ArraylndexOutOfBoundsException e) {
      System.out.println("Para utilizar el programa escriba:
      CopyFile origen destino");
      return;
    }

    // copia el archivo
    try {
      do {
        i = fin.read () ;
        if(i != -1) fout.write(i);
      } while(i!= -1);
    } catch(IOException e) {
      System.out.println("Error");
    }

    fin.close ();
    fout.close ();
  }
}
```

Observe la forma en que se gestionan los posibles errores de E/S en este programa. A diferencia de otros lenguajes de programación, incluyendo C y C++, que utilizan códigos de error para informar de los errores que se producen en los archivos, Java utiliza el mecanismo de gestión de excepciones. De esta manera no sólo se consigue un manejo de archivos más claro, también se hace más fácil diferenciar la condición de fin de un archivo de errores que se producen en la entrada. En C/C++, muchas funciones de entrada devuelven el mismo valor cuando se produce un error que cuando se llega al final del archivo, es decir, en C/C++, una condición de EOF, se transforma a menudo en el mismo valor que un error de entrada. Esto implica que el programador debe incluir sentencias adicionales en el programa para determinar cuál de los dos eventos ha ocurrido realmente. En Java, los errores se pasan al programa por medio de las excepciones, no de valores que devuelve el método **read( ).** Por tanto, cuando **read( )** devuelve -1, significa sólo una cosa: que se ha llegado al final del archivo.

Fundamentos de applets

Todos los ejemplos anteriores de este libro han sido aplicaciones Java en modo consola. Sin embargo, las aplicaciones de consola sólo constituyen una parte de los programas Java. Otro tipo de programas en Java son los applets. Como ya se comentó en el Capítulo 1, los *applets* son pequeñas aplicaciones a las que se accede en un servidor de Internet se transmiten a través de la red, se instalan automáticamente y se ejecutan como parte de un documento Web. Cuando un applet llega al cliente tiene un acceso limitado a los recursos, por lo que puede crear una interfaz de usuario multimedia y ejecutar cálculos complejos sin ningún riesgo de virus o de violación de la integridad de datos.

Muchos de los temas relacionados con la creación y uso de los applets se tratan en la Parte II, donde se examina el paquete **applet** y en la Parte III donde se describe Swing. Sin embargo, aquí se presentan los fundamentos de la creación de applets, porque éstos no se estructuran de la misma forma que los programas vistos hasta el momento. Como se verá, los app1ets se diferencian de las aplicaciones de consola en algunas áreas clave.

Comencemos con un applet sencillo:

```
import java.awt.*;
import java.applet.*;

public class SimpleApplet extends Applet {
  public void paint(Graphics g) {
    g.drawString("Un applet sencillo", 20, 20);
  }
}
```

Este applet comienza con dos sentencias **import**. La primera importa las clases del conjunto de herramientas de ventana abstracta (AWT). Los applets interactúan con el usuario a través del AWT, no a través de las clases de E/S basadas en la consola. El AWT permite desarrollar interfaces gráficas, basadas en el sistema de ventanas. Es fácil suponer que el AWT es bastante amplio y sofisticado. Por este motivo varios capítulos de la Parte II se dedican a analizarlo en detalle. Afortunadamente, en este sencillo applet se utiliza de forma muy limitada el AWT. También es posible utilizar Swing para crear la interfaz gráfica de un applet, esto se describirá más adelante. La segunda sentencia **import** importa el paquete **applet**, que contiene la clase **Applet**. Cada applet que se cree debe ser una subclase de **Applet**.

La siguiente línea del programa declara la clase **SimpleApplet**. Esta clase se debe declarar como **public**, ya que se accederá a la misma desde código externo al programa.

Dentro de **SimpleApplet**, se declara el método **paint( )**. Este método está definido por AWT y debe ser sobrescrito por el applet. Se llama al método **paint( )** cada vez que el applet debe mostrar su salida. Hay distintos motivos que darán lugar a esta situación. Por ejemplo, si la ventana en la que se está ejecutando el applet es tapada por otra ventana y después pasa otra vez a primer plano, o bien si se minimiza la ventana del applet y después se restaura. También se llama al método **paint( )** cuando comienza la ejecución del applet. El método **paint( )** tiene un parámetro del tipo **Graphics**. Este parámetro contiene el contexto gráfico que describe el entorno gráfico en el que se ejecuta el applet. Este contexto se utiliza cuando se requiere presentar la salida del applet.

Dentro de **paint( )** se llama al método **drawString( )**, que es un miembro de la clase **Graphics**. Este método imprime una cadena a partir de la posición X,Y especificada y tiene la siguiente forma general:

void drawString(String *mensaje*, int *x*, int *y*)

Donde, *mensaje* es la cadena que se imprimirá a partir de las coordenadas x, y. En una ventana de Java, la esquina superior izquierda corresponde a la posición 0,0. La llamada al método **drawString( )** en el applet da lugar a que se imprima el mensaje "Un applet sencillo" en la posición 20,20.

Observe que el applet no tiene el método **main( )**. A diferencia de los programas en Java, los applets no comienzan ejecutando el método **main( )**. En efecto, la mayoría de los applets no tienen dicho método. La ejecución de un applet comienza cuando se pasa el nombre de su clase a un visualizador de applets o a un navegador Web.

Después de capturar el código fuente de **SimpleApplet**, es posible compilarlo de la misma forma en que se han compilado los programas. Sin embargo, la ejecución de **SimpleApplet** es un proceso distinto. Existen dos formas de ejecución de un applet:

- Ejecutar el applet dentro de un navegador Web compatible con Java.
- Utilizar un visualizador de applets como **appletviewer**. Un visualizador de applets ejecuta el applet en una ventana. En general, ésta es la forma más rápida y sencilla de probar el applet.

A continuación se describe cada uno de estos dos métodos.

Para ejecutar un applet en un navegador Web, es necesario escribir un pequeño archivo de texto HTML que contiene la etiqueta apropiada para cargar al applet. Para ello SUN recomienda utilizar la etiqueta APPLET. Éste es el archivo HTML que ejecuta **SimpleApplet**:

```
<applet code="SimpleApplet" width=200 height=60>
</applet>
```

Las sentencias **width** y **height** especifican las dimensiones del área de la pantalla utilizada por el applet. (La etiqueta APPLET contiene otras opciones que se analizarán con más detalle en la Parte II). Después de crear el archivo, se ejecuta el navegador y se carga este archivo. Como resultado se ejecuta **SimpleApplet**.

Para ejecutar **SimpleApplet** con un visualizador de applets, se ejecuta también el archivo HTML anterior. Por ejemplo, si el archivo HTML se llama **RunApp**.**html**, entonces la siguiente línea de comandos servirá para ejecutar **SimpleApplet**:

```
C:\>appletviewer RunApp.html
```

Sin embargo, se puede utilizar un método mejor para realizar pruebas rápidamente. Este método consiste en incluir un comentario en la cabecera del archivo fuente de Java con la etiqueta APPLET. De esta forma, el código está documentado con un prototipo de las sentencias HTML necesarias, y se puede probar el applet compilado, iniciando simplemente el visualizador de applets con el archivo de código fuente Java. Si se utiliza este método, el archivo fuente **SimpleApplet** sería el siguiente:

```
import java.awt.*;
import java.applet.*;
/*
<applet code="SimpleApplet" width=200 height=60>
</applet>
*/

public class SimpleApplet extends Applet {
  public void paint(Graphics g) {
    g.drawString("Un applet sencillo", 20, 20);
  }
}
```

De esta forma, se pueden desarrollar rápidamente applets siguiendo estos tres pasos:

1. Editar el archivo fuente Java.
2. Compilar el programa.
3. Ejecutar el visualizador de applets, especificando el nombre del archivo fuente del applet. El visualizador encontrará la etiqueta APPLET en el comentario y ejecutará el applet.

En la siguiente ilustración de muestra la ventana generada por **SimpleApplet:**

Aunque los applets se discuten con más profundidad más adelante en este libro, los puntos clave que conviene recordar hasta este momento son:

- Los applets no necesitan un método **main( )**.
- Los applets se ejecutan con un visualizador de applets o un navegador compatible con Java.
- Para las E/S del usuario no se utilizan las clases de flujo de E/S de Java. En su lugar, se utiliza la interfaz que proporciona AWT o Swing.

Los modificadores transient y volatile

Java define dos modificadores de tipo interesantes: **transient** y **volatile**. Estos modificadores se utilizan para tratar situaciones específicas.

Cuando una variable de instancia se declara como **transient**, entonces no es necesario mantener su valor cuando el objeto se almacena. Por ejemplo:

```
class T {
  transient int a; // no persistente
  int b; // persistente
}
```

Si un objeto del tipo **T** se guarda en un área de almacenamiento persistente, el contenido de **a** no se guardaría, mientras que el de **b** sí.

El modificador **volatile** indica al compilador que la variable modificada por **volatile** se puede cambiar de forma inesperada por otras partes del programa. Una de estas situaciones está relacionada con los programas multihilos, tal y como se ha visto en un ejemplo del Capítulo 11. En ocasiones, en un programa multihilo, dos o más hilos comparten la misma variable de instancia. Por razones de eficacia, cada hilo puede guardar su propia copia de la variable compartida. La copia real o *maestra* de la variable se actualiza en diferentes instantes, como por ejemplo cuando entra en un método **synchronized**. Si bien este enfoque puede funcionar correctamente, también puede ser ineficiente en ocasiones. Lo que importa realmente es que la copia maestra de la variable refleje siempre el estado actual. Para garantizar esto, simplemente hay que especificar la variable como **volatile**, que indica al compilador que siempre debe utilizar la copia maestra de una variable **volatile** (o, por lo menos, mantener siempre actualizadas las copias privadas respecto a la maestra, y viceversa). Además, los accesos a la variable maestra se deben ejecutar en el mismo orden en que se ejecutan sobre cualquier copia privada.

instanceof

En ocasiones es útil conocer el tipo de un objeto en tiempo de ejecución. Por ejemplo, si tuviéramos un hilo de ejecución que genera varios tipos de objetos, y otro hilo que procesa esos objetos. En esta situación, es interesante para el hilo que procesa conocer el tipo de cada objeto cuando lo recibe. Otra situación en la que es importante conocer el tipo de un objeto en tiempo de ejecución es la de la conversión de tipos. En Java, una conversión inválida da lugar a un error en tiempo de ejecución. En el tiempo de compilación es factible detectar muchas conversiones inválidas, sin embargo, las conversiones en las que están implicadas jerarquías de clases pueden dar lugar a conversiones inválidas que sólo se pueden detectar en la ejecución. Por ejemplo, imaginemos una superclase, denominada A que produce dos subclases denominadas B y C. Se puede convertir un objeto del tipo B al tipo A, o convertir un objeto del tipo C al tipo A, pero no está permitido convertir un objeto del tipo B al tipo C o viceversa. Como un objeto del tipo A puede referirse a objetos del tipo B o C, ¿cómo se puede saber, durante la ejecución, qué tipo de objeto es el que realmente está siendo referenciado antes de intentar la conversión al tipo C? Podría ser un objeto de los tipos A, B o C. Si es del tipo B, se lanzará una excepción en tiempo de ejecución. Java proporciona el operador **instanceof** para responder a esta cuestión.

El operador **instanceof** tiene la siguiente forma general:

objeto instanceof *tipo*

donde *objeto* es una instancia de una clase, y *tipo* es una clase. Si *objeto* es del tipo especificado o se puede convertir en ese tipo, entonces el operador **instanceof** dará como resultado el valor **true**. En caso contrario, su resultado es **false**. Por lo tanto, **instanceof** es el medio por el cual el programa puede obtener información sobre un objeto en tiempo de ejecución.

El siguiente programa muestra la utilización del operador **instanceof**:

```
// Ejemplo del operador instanceof.
class A {
  int i, j;
}

class B {
  int i, j;
}

class C extends A {
  int k;
}

class D extends A {
  int k;
}

class InstanceOf {
  public static void main(String args[]) {
    A a = new A();
    B b = new B();
    C c = new C();
    D d = new D();
```

```
    if(a instanceof A)
      System.out.println("a es una instancia de A");
    if(b instanceof B)
      System.out.println("b es una instancia de B");
    if{c instanceof C)
      System.out.println{"c es una instancia de C");
    if(e instanceof A)
      System.out.println{"c se puede convertir a A");

    if(a instanceof C)
      System.out.println{"a se puede convertir a C");

    System.out.println();

    // Comparar los tipos de tipos derivados
    A ob;

    ob = d; // una referencia a d de tipo A
    System.out.println("ob hace referencia a d");
    if(ob instanceof D)
      System.out.println("ob es una instancia de D");

    System.out.println();

    ob = c; // una referencia a c de tipo A
    System.out.println("ob hace referencia a c");

    if(ob instanceof D)
      System.out.println("ob se puede convertir a D");
    else
      System.out.println("ob no se puede convertir a D");

    if(ob instanceof A)
      System.out.println("ob se puede convertir a A");

    System.out.println();

    // todos los objetos se pueden convertir en Object
    if(a instanceof Object)
      System.out.println("a se puede convertir a Object");
    if(b instanceof Object)
      System.out.println("b se puede convertir a Object");
    if(c instanceof Object)
      System.out.println("c se puede convertir a Object");
    if(d instanceof Object)
      System.out.println("d se puede convertir a Object");
  }
}
```

La salida de este programa es la siguiente:

```
a es una instancia de A
b es una instancia de B

c es una instancia de C
c se puede convertir a A
```

```
ob hace referencia a d
ob es una instancia de D

ob hace referencia a c
ob no se puede convertir a D
ob se puede convertir a A

a se puede convertir a Object
b se puede convertir a Object
c se puede convertir a Object
d se puede convertir a Object
```

El operador **instanceof** no se necesita en la mayoría de programas, ya que, generalmente, se conoce el tipo de objetos con los que se está trabajando. Sin embargo, puede ser muy útil cuando se escriben rutinas generalizadas que operan sobre objetos de una jerarquía de clases compleja.

strictfp

Java 2 ha añadido una palabra clave nueva al lenguaje Java, denominada **strictfp**. Con la creación de Java 2, el modelo de cálculo en punto flotante se ha relajado ligeramente. Específicamente, el nuevo modelo no requiere el truncamiento de ciertos valores intermedios que se producen durante los cálculos. Modificando una clase o método con la palabra clave **strictfp**, se asegura que los cálculos en punto flotante, y por tanto todos los truncamientos, se efectúen del mismo modo que en las versiones anteriores de Java. Cuando se modifica una clase con **strictfp**, automáticamente se modifican todos los métodos de la clase con **strictfp**.

El siguiente fragmento, por ejemplo, indica a Java que utilice el modelo original de punto flotante para los cálculos en todos los métodos definidos en **MiClase**:

```
strictfp class MiClase { //...
```

Muchos programadores no utilizan nunca **strictfp**, porque afecta únicamente a un pequeño grupo de problemas.

Métodos nativos

Aunque poco frecuente, ocasionalmente puede ser necesario llamar a una subrutina escrita en otro lenguaje distinto de Java. Normalmente, esa subrutina existe como un código ejecutable para la CPU y el entorno de trabajo, es decir, código nativo. Por ejemplo, puede ser conveniente llamar a una subrutina de código nativo para lograr un tiempo de ejecución más rápido, o bien puede ser necesario utilizar una biblioteca especializada, como un paquete estadístico. Sin embargo, como los programas Java se compilan en un código binario que es interpretado después por el intérprete Java, podría parecer imposible llamar a una subrutina de código nativo desde un programa Java. Afortunadamente, esta conclusión es falsa.

Java facilita la palabra clave **native** que se utiliza para declarar métodos de código nativo. Una vez declarados, se puede llamar a estos métodos desde el programa Java del mismo modo que se llama cualquier otro método de Java.

Para declarar un método nativo, se coloca el nombre del método precedido por el modificador **native**, pero no se define ningún cuerpo para el método. Por ejemplo:

```
public native int meth ();
```

Después de declarar un método nativo, se debe escribir el método y seguir una serie compleja de pasos para enlazado con el código Java.

La mayor parte de los métodos nativos están escritos en C. El mecanismo que se utiliza para integrar código C con programas Java se denomina Interfaz Nativa de Java (JNI, por sus siglas en inglés, *Java Native Interface*). Una descripción detallada de la INI está más allá de los propósitos de este libro, pero la siguiente descripción proporciona información suficiente en la mayoría de las aplicaciones.

NOTA *Los pasos precisos que se han de seguir varían según las diferentes versiones y entornos de Java. También dependen del lenguaje en el que esté implementado el código nativo. La siguiente discusión considera un entorno Windows. El lenguaje en el que se implementa el método nativo es C.*

La manera más fácil de entender el proceso es por medio de un ejemplo. Para comenzar veamos un programa corto que utiliza un método **nativo** denominado **test( )**:

```
// Un ejemplo sencillo que utiliza métodos nativos.
public class NativeDemo {
  int i;
  public static void main(String args[]) {
    NativeDemo ob = new NativeDemo();

    ob.i = 10;
    System.out.println("Esto es ob.i antes del método nativo:" +
                       ob.i) ;
    ob.test(); // llamada a un método nativo
    System.out.println("Esto es ob.i después del método nativo:" +
                       ob.i);
  }
  // Declaración del método nativo
  public native void test();

  // carga la DLL que contiene el método estático
  static{
    System.loadLibrary("NativeDemo");
  }
}
```

Observe que el método **test( )** se declara como **native** y no tiene cuerpo. Este método será implementado en C. Observe también el bloque **static**. Como se ha explicado anteriormente en este libro, un bloque **static** se ejecuta una sola vez, cuando el programa comienza la ejecución o, más precisamente, cuando se carga por primera vez su clase. En este caso, se utiliza para cargar la biblioteca de enlace dinámico (DLL) que contiene la implementación nativa de **test( )**. Posteriormente se verá cómo se crea esta biblioteca.

La biblioteca se carga utilizando el método **loadLibrary( )**, que es parte de la clase **System**. Su forma general es:

static void loadLibrary(String *nombreArchivo*)

Donde *nombreArchivo* es una cadena que especifica el nombre del archivo que contiene la biblioteca. En el entorno Windows se supone que este archivo tiene la extensión. DLL.

Cuando se compila el programa Java, se genera el archivo **NativeDemo.cIass.** A continuación se debe utilizar **javah.exe** para generar el archivo: **NativeDemo.h.** (**javah.exe** está incluido en JDK.) En la implementación de **test( )** habrá que incluir el archivo **NativeDemo.h.** Para generar este archivo se utiliza el siguiente comando:

javah -jni NativeDemo

Este comando genera un archivo cabecera denominado **NativeDemo.h.** Este archivo debe incluirse en el archivo C que implementa **test( ).** La salida generada por este comando es la siguiente:

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class NativeDemo */

#ifndef _Inc1uded_NativeDemo
#define _Included_NativeDemo
#ifdef _ _cplusplus
extern "C" {
#endif
/*
 * Class: NativeDemo
 * Method: test
 * Signature: ()V
 */
JNIEXPORT void JNlCALL Java_NativeDemo_test
  (JNIEnv *, jobject);

#ifdef _ _cplusplus
}
#endif
#endif
```

Prestemos especial atención a la siguiente línea, que define el prototipo para la función **test( ):**

```
JNIEXPORT void JNICALL Java_NativeDemo_test(JNIEnv *, jobject);
```

Observe que el nombre de la función es **Java_NativeDemo_test( ).** Éste es el nombre que se debe utilizar para la función, es decir, en lugar de crear una función C denominada **test( )**, se creará una función denominada **Java_NativeDemo_test( ).** La componente **NativeDemo** del prefijo se añade porque identifica el método **test( )** como parte de la clase **NativeDemo**. Recuerde que otra clase puede definir su propio método nativo **test( )** completamente diferente del declarado por **NativeDemo**. Al incluir el nombre de la clase en el prefijo es posible distinguir distintas versiones del método. Como regla general, las funciones nativas tendrán un nombre cuyo prefijo incluya el nombre de la clase en la que se declaran.

Después de generar el archivo de cabecera necesario, se escribe la implementación de **test( )** y se almacena en un archivo denominado **NativeDemo.c:**

```
/*  Este archivo contiene la versión C
    del método test().
*/
```

```
#include <jni.h>
#include "NativeDemo.h"
#include <stdio.h>

JINIEXPORT void JNlCALL Java_NativeDemo_test(JNIEnv *env, jobject obj)
{
  jclass cls;
  jfieldID fid;
  jint i;

  printf("Comienzo del método nativo.\n");
  cls = (*env)->GetObjectClass(env, obj);
  fid = (*env)->GetFieldID(env, cls, "i", "I");

  if(fid == 0) {
    printf ("No se puede obtener el id del campo. \n");
    return;
  }
  i = (*env)->GetIntField(env, obj, fid);
  printf("i = %d\n" , i);
  (*env)->SetIntField(env, obj, fid, 2*i);
  printf("Final del método nativo.\n");
}
```

Observe que este archivo incluye **jni.h**, que contiene la información de la interfaz. Este archivo lo proporciona el compilador Java. El archivo cabecera **NativeDemo**.**h** fue creado anteriormente por **javah.**

En esta función, el método **GetObjectClass( )** se usa para obtener una estructura C con información sobre la clase **NativeDemo**. El método **GetFieldID( )** devuelve una estructura C con información relativa al campo llamado "i" de la clase. **GetIntField( )** recupera el valor original de ese campo. **SetIntField( )** almacena un valor actualizado en ese campo. (Para otros métodos que manejen otros tipos de datos, véase el archivo **jni.h.**)

Después de crear **NativeDemo.c**, se debe compilar para obtener el correspondiente archivo DLL. Para ello, con el compilador de C/C++ de Microsoft, se utiliza la siguiente línea de comandos (será necesario especificar la ubicación de los archivos **jni.h** y su archivo complementario **jni_md.h**):

```
C1 /LD NativeDemo.c
```

Esto genera un archivo denominado **NativeDemo.dll**. Una vez hecho todo esto, se puede ejecutar el programa Java, obteniendo la siguiente salida:

```
Esto es ob.i antes del método nativo: 10
Comienzo del método nativo.
i = 10
Final del método nativo.
Esto es ob.i después del método nativo: 20
```

Problemas con los métodos nativos

Los métodos nativos parecen ofrecer muchas posibilidades, porque permiten tanto acceder a bibliotecas de rutinas existentes como conseguir tiempos de ejecución más rápidos. Sin embargo, introducen dos problemas significativos:

- **Riesgos potenciales para la seguridad**: Dado que un método nativo ejecuta realmente código máquina, puede acceder a cualquier parte del sistema local; es decir, un código nativo no está confinado al entorno de ejecución de Java. Esto podría permitir la entrada de virus, por ejemplo. Por este motivo, los applets no pueden usar métodos nativos. Además, la carga de archivos DLL puede estar restringida en el sistema y sujeta a la aprobación del administrador.
- Pérdida **de portabilidad**: Dado que el código nativo está contenido en una DLL, debe estar presente en la máquina que ejecuta el programa Java. Más aún, como cada código nativo depende del CPU y del sistema operativo, cada DLL es intrínsecamente no portable. Por tanto, una aplicación Java que utilice métodos nativos sólo se podrá ejecutar en una máquina compatible con la DLL que ha sido instalada.

El uso de métodos nativos se debe restringir, debido a que aportan a los programas Java un riesgo de seguridad, y también afectan a la portabilidad.

assert

Otra adición relativamente nueva en Java es la palabra clave **assert**. Ésta es utilizada durante el desarrollo de un programa para crear aserciones. Una aserción es una condición que debe ser cierta durante la ejecución del programa. Por ejemplo, podríamos tener un método que debería regresar siempre un número positivo. Esto puede ser verificado realizando la aserción respecto a que el valor regresado por el método debe ser mayor a cero utilizando una sentencia **assert**. En tiempo de ejecución si la condición se cumple no ocurre ninguna acción particular; sin embargo, si la condición es falsa se lanza un error del tipo **AssertionError**. Las aserciones son utilizadas comúnmente durante la fase de pruebas para verificar que alguna condición necesaria se cumpla. Prácticamente no se utilizan en el código terminado.

La palabra clave **assert** tiene dos formas, la primera es la siguiente:

assert *condición*;

Donde *condición* es una expresión que produce un resultado del tipo boolean. Si el resultado es true, la aserción se satisface y por tanto no se realiza ninguna acción. Si condición es falsa, entonces la aserción ha fallado y se lanza un objeto **AssertionError**.

La segunda forma de **assert** es:

assert *condición* : *expresión*;

En esta versión, *expresión* es un valor que es enviado al constructor del objeto tipo **AssertionError**. Este valor es convertido a String y mostrado cuando el objeto es lanzado si la aserción no se satisface. Comúnmente, esta expresión es directamente una cadena de caracteres, sin embargo cualquier expresión de tipo diferente a **void** está permitida siempre que ésta pueda ser convertida a cadena.

El siguiente ejemplo utiliza **assert** para verificar que el valor regresado por el método **getnum( )** sea positivo.

```
// Ejemplo de assert
class AssertDemo {
  static int val = 3;

  // el método regresa un valor entero
  static int getnum() {
    return val--;
  }

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

    for(int i=0; i < 10; i++) {
      n = getnum();

      assert n > 0; // fallará cuando n tenga el valor 0

      System.out.println("n es" + n);
    }
  }
}
```

Para activar la revisión de aserciones en tiempo de ejecución es necesario especificar la opción **–ea** en la ejecución de Java. Por ejemplo, para activar la revisión de aserciones para **AssertDemo** se escribe:

```
java –ea AssertDemo
```

Después de compilar y ejecutar el programa se obtiene la siguiente salida:

```
n es 3
n es 2
n es 1
Exception in thread "main" java.lang.AssertionError
       At AssertionDemo.main(AssertDemo.java:17)
```

En **main( )** se realizan repetidamente llamadas al método **getnum( )**, las cuales retornan un valor entero. El valor retornado por **getnum( )** es asignado a la variable **n** y luego verificado utilizando la sentencia **assert**:

```
assert n > 0; // fallará cuando n tenga el valor 0
```

Esta sentencia fallará cuando **n** sea igual a 0, lo cual ocurre en la cuarta llamada al método. Cuando esto ocurre se lanza una excepción.

Como se explicó antes, es posible especificar el mensaje a mostrar cuanto la aserción no se cumple. Por ejemplo, si en el programa anterior sustituimos la sentencia de aserción por la siguiente:

```
assert n > 0 : "¡n es un valor negativo!";
```

El programa generaría la siguiente salida:

```
n es 3
```

```
    n es 2
    n es 1
    Exception in thread "main" java.lang.AssertionError: ¡n es
    un valor negativo!
                    At AssertDemo.main(AssertDemo.java:17)
```

Un punto importante a tomar en cuenta sobre las aserciones es el hecho de que no deben ser utilizadas para que realicen acciones requeridas por el programa. Esto debido a que los programas son comúnmente ejecutados por Java con revisión de aserciones desactivadas. Por ejemplo, consideremos el siguiente programa:

```
// Una manera equivocada de utilizar assert
class AssertDemo {
    // generador de números aleatorios
    static int val = 3;

    // regresa un entero
    static int getnum() {
      return val--;
    }

    public static void main(String args[])
    {
      int n = 0;

      for(int i = 0; i < 10; i++) {

      assert (n = getnum()) > 0; // ¡ésta no es una buena idea!

      System.out.println("n es " + n);
    }
  }
}
```

En esta versión del programa la llamada a **getnum( )** se movió dentro de la sentencia **assert**. Aunque este código funciona correctamente cuando la revisión de aserciones es activada, este código no funcionará cuando la revisión de aserciones no esté activa porque la llamada a **getnum( )** nunca será ejecutada.

Las aserciones son una buena adición a Java debido, principalmente, a que hacen más eficiente la revisión de errores durante el desarrollo de software. Por ejemplo, antes de la existencia de las **aserciones**, si se deseaba verificar que **n** contuviera un valor positivo se debía utilizar un código como éste:

```
if(n < 0) {
    System.out.println("n es un valor negativo");
    return; // regresar o bien lanzar una excepción
}
```

Con **aserciones** sólo necesitamos una línea de código. Además no es necesario eliminar las sentencias **assert** en el código terminado.

Opciones para activar y desactivar la aserción

Al ejecutar programas, es posible inactivar las aserciones utilizando la opción **–da.** Para activar o inactivar un paquete específico se utiliza la opción **–ea** o **–da** seguido del nombre del paquete. Por ejemplo para activar la revisión de aserciones en un paquete llamado **MiPaquete**, escribiríamos:

```
-ea:MiPaquete
```

Para inactivar la verificación de aserciones en el paquete MiPaquete, escribiríamos:

```
-da:MiPaquete
```

Para activar o inactivar todos los subpaquetes de un paquete dado, se escribe el nombre del paquete seguido de tres puntos. Por ejemplo:

```
- ea:MiPaquete...
```

También es posible especificar una clase con las opciones **–ea** y **–da**. Por ejemplo, esto activa la revisión de aserciones en la clase **AssertDemo**:

```
-ea:AssertDemo
```

Importación estática de clases e interfaces

JDK 5 añade una nueva característica a Java denominada *importación estática* que extiende las capacidades de la palabra clave **import**. Al combinar las palabras clave **import** y **static** obtenemos un **import** que puede emplearse para importar los miembros estáticos de una clase o interfaz. Cuando se utiliza importación estática, es posible acceder a los miembros estáticos de manera directa mediante sus nombres, sin tener que incluir el nombre de la clase. Esto simplifica y reduce la sintaxis requerida para utilizar miembros estáticos.

Para entender la utilidad de una importación estática veamos primero un ejemplo que *no* la utiliza. El siguiente programa calcula la hipotenusa de un triángulo rectángulo. Este código utiliza dos métodos estáticos de las bibliotecas de Java, específicamente de la clase **Math**, la cual pertenece al paquete **java.lang**. El primer método es **Math.pow( )**, regresa el valor de un número elevado a una determinada potencia. El segundo método es **Math.sqrt( )**, que regresa la raíz cuadrada del valor recibido como argumento.

```
// Calcula la hipotenusa de un triángulo rectángulo
class Hypot {
  public static void main(String args[]) {
    double sidel, side2;
    double hypot;

    sidel = 3.0;
    side2 = 4.0;

    // observe como sqrt() y pow() deben estar
    // precedidos por el nombre de la clase Math
```

```
    hypot = Math.sqrt(Math.pow(sidel, 2) +
                      Math.pow(side2, 2));

    System.out.println("Con catetos de longitud " +
                       sidel + " y " + side2 +
                       " la hipotenusa es " +
                       hypot) ;
  }
}
```

Dado que **pow( )** y **sqrt( )** son métodos estáticos, deben ser llamados utilizando el nombre de la clase que los contiene, ésa es la clase **Math**. Eso origina que la ecuación de cálculo de hipotenusa se escriba como

```
hypot = Math.sqrt(Math.pow(sidel, 2) +
                  Math.pow(side2, 2));
```

Como lo ilustra este ejemplo, tener que especificar el nombre de la clase cada vez que **pow( )**, **sqrt( )** o cualquier otro método de la clase Math de Java, como **sin( )**, **cos( )** y **tan( )**, son llamados puede resultar tedioso.

Para eliminar la necesidad de incluir el nombre de la clase en este tipo de invocaciones, Java provee el concepto de importación estática. El siguiente código es el mismo programa anterior pero aplicando importación estática.

```
// Ejemplo de importación estática con los métodos pow y sqrt
import static java.lang.Math.sqrt;
import static java.lang.Math.pow;

// Calcular la hipotenusa de un triángulo rectángulo
class Hypot {
  public static void main(String args[]) {
    double sidel, side2;
    double hypot;

    sidel = 3.0;
    side2 = 4.0;

    // Aquí son llamados los métodos sqrt() y pow()
    // sin necesitad de anteponer el nombre de la clase Math
    hypot = sqrt(pow(sidel, 2) + pow(side2, 2));

    System.out.println("Con catetos de longitud " +
                       sidel + " y " + side2 +
                       " la hipotenusa es " +
                       hypot) ;
  }
}
```

En esta versión, los nombres de los métodos **sqrt** y **pow** están disponibles gracias a las sentencias:

```
import static java.lang.Math.sqrt;
import static java.lang.Math.pow;
```

Con esto ya no es necesario colocar el nombre de la clase Math antes de los métodos **sqrt( )** y **pow( )**. Así, el cálculo de la hipotenusa puede ser realizado con la siguiente línea:

```
hypot = sqrt(pow(sidel, 2) + pow(side2, 2));
```

Esta forma tiene visiblemente una mejor legibilidad.

Existen dos formas para las sentencias **import static**. La primera, utilizada en el ejemplo anterior, se utiliza para importar un miembro específico y su forma general es:

import static *paquete.nombreTipo.nombreMiembroEstatico*;

Aquí, *nombreTipo* es el nombre de la clase o interfaz que contiene al miembro estático. El nombre completo del paquete está especificado por *paquete*. El nombre del miembro está especificado por *nombreMiembroEstatico.*

La segunda forma permite importar todos los miembros estáticos de la clase o interfaz y su forma general es:

import static *paquete.nombreTipo.**;

Si se planea utilizar varios de los métodos o variables estáticos definidos en la clase entonces esta forma nos permite tener acceso a todos ellos sin tener que especificar un **import** para cada uno de manera individual. Así, el programa anterior pudo haber utilizado únicamente la siguiente sentencia **import** para permitir el acceso a los métodos **pow( )** y **sqrt( )** (y también a todos los demás miembros estáticos de la clase **Math**):

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

La importación estática no se limita a la clase **Math** y tampoco se limita sólo a métodos. Por ejemplo, la siguiente sentencia importa el campo estático **out** de la clase **System**:

```
import static java.lang.System.out;
```

Después de la sentencia anterior es posible enviar datos a la pantalla utilizando **out** sin precederla del nombre de su clase:

```
out.println("Después de importar System.out, se puede utilizar out
            directamente");
```

Si bien importar **System.out**, como se muestra en el ejemplo anterior, permite escribir sentencias más cortas, su uso no es del todo una buena idea, ya que al mismo tiempo resta claridad al código. Al leer la sentencia anterior, no es evidente de forma instantánea que out se refiere a **System.out**.

De la misma forma en que se importan los miembros estáticos de clases e interfaces definidos en el API de Java, es posible también importar los miembros estáticos de clase e interfaces propias.

Es importante no abusar de las facilidades que brinda la importación estática. Debemos recordar que la razón por la cual Java organiza sus bibliotecas en paquetes es eliminar la colisión de nombres. Cuando se importan miembros estáticos sus nombres se están incluyendo en el espacio de nombres global. De esta manera, se incrementa el riesgo de conflictos en el espacio de nombres y de ocultar nombres de forma inadvertida. Si se utiliza sólo una o dos veces un miembro estático en el programa es mejor no importarlo. Además ciertos nombres

estáticos, como **System.out**, son tan conocidos que no es recomendable importarlos. La importación estática está diseñada para ser empleada en aquellas situaciones en las cuales se utiliza repetidamente un miembro estático, como cuando se realizan cálculos matemáticos con métodos de la clase Math. En esencia esta característica debe ser utilizada cuidando no abusar de ella.

Invocación de constructores sobrecargados con la palabra clave this( )

Cuando se trabaja con constructores sobrecargados en ocasiones es útil para un constructor invocar a otro. En Java, esto se logra utilizando la palabra clave **this**. Como sigue:

this(*parámetros*)

Cuando se ejecuta **this( )**, el constructor sobrecargado cuya lista de parámetros coincide con la *lista de parámetros* especificada por parámetros es ejecutado. Luego se ejecutan las sentencias del método constructor que realizó la llamada a **this( )**. La llamada a **this( )** debe estar colocada como primer línea dentro de un constructor.

Para entender cómo se utiliza **this( )**, veamos un pequeño ejemplo. Primero, considere la siguiente clase que *no* utiliza **this( ):**

```
class MyClass
  int a;
  int b;

  // inicializa a y b individualmente
  MyClass(int i, int j) {
    a = i;
    b = j;
  }

  // inicializa a y b con el mismo valor
  MyClass(int i) {
    a = i;
    b = i;
  }

  // inicializa a y b con el valor cero
  MyClass( ) {
    a = 0;
    b = 0;
  }
}
```

Esta clase contiene tres constructores, cada uno de los cuales inicializa los valores de **a** y **b**. El primero permite pasar valores individuales para **a** y **b**. El segundo permite pasar solo un valor, el cual es asignado por igual a **a** y **b**. El tercero da a **a** y **b** el valor por omisión de cero. Utilizando **this( )** es posible reescribir **MyClass** como se muestra a continuación:

```
class MyClass {
  int a;
  int b;

  // inicializar a y b individualmente
  MyClass(int i, int j) {
    a = i;
    b = j;
  }

  // inicializar a y b con el mismo valor
  MyClass(int i) {
    this(i, i); // llama a MyClass(i, i)
  }

  // inicializa a y b con el valor cero
  MyClass( ) {
    this(0); // llama a MyClass(0);
  }
}
```

En esta versión de **MyClass**, el único constructor que asigna valores a los campos **a** y **b** es **MyClass(int, int)**, los otros dos constructores simplemente invocan, directa o indirectamente a este constructor vía **this( )**. Por ejemplo, considere lo que ocurre cuando se ejecuta la siguiente sentencia:

```
MyClass mc = new MyClass(8);
```

La llamada a **MyClass(8)** ocasiona una llamada a **this(8, 8)**, la cual a su vez se convierte en una llamada a **MyClass(8, 8)** que es el constructor de la clase **MyClass** que coincide con la lista de parámetros listados en la invocación a **this( )**. Ahora veamos que ocurre con la siguiente sentencia, la cual utiliza el constructor sin parámetros:

```
MyClass me2 = new MyClass();
```

En este caso, se llama a **this(0)**, esa llamada se convierte en una llamada a **MyClass(0)**. La llamada a **MyClass(0)** a su vez desencadena una llamada a **MyClass(0,0)**.

La invocación de constructores sobrecargados utilizando **this( )** previene la duplicación innecesaria de código. Reducir el código duplicado, en muchos casos, permite generar un código objeto más pequeño lo que a su vez causa que el tiempo que le toma a la máquina virtual cargar el código a memoria sea menor. Esto es especialmente importante para programas que han de ser distribuidos por Internet. El uso de **this( )** nos ayuda además a estructurar el programa cuando los constructores contienen grandes cantidades de código repetido.

Sin embargo es recomendable ser cuidadosos. Los constructores que llaman a **this( )** serán ejecutados más lentamente que aquellos que contienen su propio código. Esto debido a que el mecanismo de llamada y retorno entre métodos constructores añade tiempo adicional al proceso. Si sólo se van a crear algunos objetos de la clase o bien si el constructor en la clase que llama a **this( )** será utilizado en pocas ocasiones entonces el incremento en el tiempo de ejecución será insignificante. Sin embargo, si se planea crear un gran número de objetos de la

clase (en el orden de miles de objetos) durante la ejecución del programa, entonces el efecto en el rendimiento del programa será considerable. Dado que la creación de objetos afecta a todos los usuarios de la clase debemos, en muchos casos, ser cuidadosos y colocar en la balanza si necesitamos rapidez en el tiempo de carga del programa (código pequeño) aún a pesar de incrementar el tiempo requerido para la creación de un objeto.

Una consideración adicional a tomar en cuenta es que para constructores muy pequeños, como los utilizados en la clase **MyClass**, existe muy poca diferencia en el tamaño del código utilizando o no **this( )**. Actualmente, existen casos donde no se genera ninguna reducción en el tamaño del código. Esto debido al bytecode que se añade al código objeto para representar la llamada y retorno de una función.

Por ello, en ese tipo de situaciones, aun cuando se elimina el código duplicado, **this( )** no proporciona ahorro en el tiempo de carga. Sin embargo, si se añade costo que se añada en términos de más tiempo requerido para construir objetos de la clase. De ahí que **this( )** se aplique sólo a constructores que contienen una gran cantidad de código y no a aquellos que simplemente inicializan el valor de un pequeño número de variables.

Existen dos restricciones que se deben tener en cuenta cuando se utiliza **this( )**. Primero, no podemos utilizar ninguna variable de instancia de la clase del constructor en la llamada a **this( )**. Y segundo, no podemos utilizar **super( )** y **this( )** en el mismo constructor debido a que ambas sentencias están definidas para ser las primeras en ejecutarse en el constructor.

14 CAPÍTULO

Tipos parametrizados

Desde la versión original 1.0 de 1995, muchas nuevas características han sido agregadas a Java. La que ha tenido el efecto más profundo es la *genérica*, esto es los tipos parametrizados. Introducidos en el JDK 5, los tipos parametrizados han cambiado Java en dos formas muy importantes. Primero, agregaron un nuevo elemento sintáctico al lenguaje. Segundo, causó cambios a muchas de las clases y métodos en el núcleo de la API de Java. Puesto que los tipos parametrizados representan un gran cambio en el lenguaje, algunos programadores estuvieron renuentes a adoptar su uso. Sin embargo, con la versión de JDK 6, los tipos parametrizados no pueden ser ignorados. En pocas palabras, si se va a programar con Java SE 6, se van a estar utilizando tipos parametrizados constantemente. Afortunadamente, los tipos parametrizados no son difíciles de utilizar y proveen beneficios significativos para los programadores en Java.

A través del uso de tipos parametrizados, es posible crear clases, interfaces y métodos que trabajarán de forma segura con varios tipos de datos. Muchos algoritmos son lógicamente los mismos sin importar a qué tipo de datos estén siendo aplicados. Por ejemplo, el mecanismo que soporta una pila es el mismo si la pila almacena datos de tipo **Integer**, **String**, **Object** o **Thread**. Con los tipos parametrizados, se puede definir un algoritmo independientemente del tipo de datos, y luego aplicar dicho algoritmo a una amplia variedad de tipos de datos sin esfuerzo adicional. El poder expresivo que los tipos parametrizados agregan al lenguaje cambia fundamentalmente la forma en que se escribe el código de Java.

Quizá la característica de Java que ha sido más significativamente afectada por los tipos parametrizados es la *Estructura de Colecciones*. La Estructura de Colecciones es parte de la API de Java y se describe a detalle en el Capítulo 17, sin embargo vale la pena hablar un poco de ella ahora. Una *colección* es un grupo de objetos. La Estructura de Colecciones define muchas clases, tales como listas y mapas, que administran las colecciones. Las clases que representan colecciones siempre han sido capaces de trabajar con cualquier tipo de objeto. El beneficio que los tipos parametrizados agregan, es que las clases de colección ahora pueden ser utilizadas con completa seguridad. Así, además de proveer un nuevo elemento poderoso al lenguaje, los tipos parametrizados también habilitan una característica existente que ha sido sustancialmente mejorada. Ésta es la razón por la cual los tipos parametrizados representan una importante extensión a Java.

Este capítulo describe la sintaxis, teoría y uso de los tipos parametrizados. También muestra como los tipos parametrizados proveen seguridad al manejar tipos en casos que, anteriormente,

eran complicados. Una vez que se haya completado este capítulo, seguramente el lector deseará examinar el Capítulo 17 que habla de la Estructura de Colecciones. En el Capítulo 17 se encuentran muchos ejemplos funcionando de tipos parametrizados.

NOTA *Los tipos parametrizados fueron agregados en JDK 5. El código fuente que utiliza tipos parametrizados no puede ser compilado por versiones de* ***javac*** *anteriores a la versión 5.*

¿Qué son los tipos parametrizados?

El término *tipos parametrizados* es el núcleo de la *característica genérica*. Los tipos parametrizados son importantes porque proporcionan la habilidad de crear clases, interfaces y métodos en los cuales los tipos de datos sobre los cuales operan son especificados como parámetros. Utilizando tipos parametrizados, es posible crear una clase, por ejemplo, que automáticamente funcione con diferentes tipos de datos. Una clase, una interfaz, o un método que opere sobre un tipo parametrizado es llamada *genérica*, de ahí que hablemos de *clases genéricas* o *método genéricos*.

Es importante entender que Java siempre ha contado con la habilidad de crear clases, interfaces y métodos generalizados gracias al uso de referencias a objetos de tipo **Object**. Dado que **Object** es la superclase de todas las otras clases, una referencia de tipo **Object** se puede referir a cualquier otro tipo de objeto. Así, en el código previo a la existencia de tipos parametrizados, clases, interfaces y métodos generalizados, utilizaban referencias a **Object** para operar sobre varios tipos de objetos. El problema era la falta de seguridad en el manejo de los tipos de datos.

Los tipos parametrizados agregan la seguridad que hacia falta. Además estilizan el proceso ya que evitan que sea necesario realizar la conversión explicita de tipos para traducir entre **Object** y el tipo de datos sobre el que se requiere trabajar. Con los tipos parametrizados, todas las conversiones de tipos son automáticas e implícitas. Por ello, los tipos parametrizados expanden la capacidad de reutilizar código y permite hacerlo de una forma más fácil y segura.

NOTA *Una advertencia para los programadores de C++: Aunque los tipos parametrizados son similares a las plantillas de C++, no son lo mismo. Existen algunas diferencias fundamentales entre los dos enfoques. Si se tiene experiencia en C++, es importante no sacar conclusiones apresuradas sobre cómo funcionan los tipos parametrizados.*

Un ejemplo sencillo con tipos parametrizados

Comencemos con un ejemplo simple de una clase genérica. El siguiente programa define dos clases. La primera es la clase genérica llamada **Gen** y la segunda es la clase **GenDemo** que utiliza a **Gen**.

```
// Un ejemplo de clase genérica
// Donde, T es un parámetro de tipo que
// será reemplazado por un tipo real
// cuando un objeto de tipo Gen sea creando
class Gen<T> {
  T ob; // declara un objeto de tipo T
```

```
  // Pasa al constructor una referencia a
  // un objeto de tipo T
  Gen(T o) {
    ob = o;
  }

  // Devuelve ob.
  T getob( ) {
    return ob;
  }

  // Muestra el tipo de T.
  void showType( ) {
    System.out.println("Tipo de T es" +
                       ob.getClass( ).getName( ));
  }
}

// Esta clase utiliza a la clase con tipos parametrizados.
class GenDemo {
  public static void main(String args[]) {
    // Crea una referencia a Gen para objetos Integer.
    Gen<Integer> iOb;

    // Crea un objeto Gen<Integer> y asigna su
    // referencia a iOb. Nótese el uso de autoboxing
    // para encapsular el valor 88 dentro de un objeto Integer
    iOb = new Gen<Integer> (88);

    // Muestra el tipo de datos utilizado por iOb.
    iOb.showType ( );

    // Obtiene el valor en iOb. Nótese que
    // no se necesita hacer la conversión explícita de tipos
    int v = iOb.getob( );
    System.out.println("valor: " + v);

    System.out.println( );

    // Crea un objeto Gen para valores de tipo String
    Gen<String> strOb = new Gen<String> ("Prueba de Tipos Parametrizados");

    // Muestra el tipo de dato utilizado por strOb.
    strOb.showType( );

    // Obtiene el valor de strOb. De nuevo, nótese
    // que no se necesita hacer la conversión explícita de tipos
    String str = strOb.getob( );
    System.out.println("valor: " + str);
  }
}
```

La salida producida por el programa es la siguiente:

```
Tipo de T es java.lang.Integer
valor: 88
```

```
Tipo de T es java.lang.String
valor: Prueba de Tipos Parametrizados
```

Examinemos el programa con más detenimiento.

En primer lugar, observemos como la clase **Gen** es declarada en la siguiente línea:

```
class Gen<T> {
```

Donde **T** es el nombre de un *parámetro de tipo*. Este nombre se utiliza como un marcador de posición para el verdadero tipo que será pasado a **Gen** cuando un objeto sea creado. Así, **T** es utilizado dentro de **Gen** donde sea que el tipo parametrizado sea requerido. Note que **T** está colocado dentro de **< >**. Esta sintaxis puede ser generalizada. En cualquier momento que se requiera declarar un tipo parametrizado, éste se especifica dentro de paréntesis angulares. Dado que la clase **Gen** utiliza un *tipo parametrizado*, **Gen** es una clase genérica.

A continuación, **T** es utilizada para declarar un objeto llamado **ob**, como se muestra a continuación:

```
T ob; //declara un objeto de tipo T
```

Como se explicó, **T** es un marcador de posición para el tipo actual que será especificado cuando un objeto **Gen** sea creado. Así, **ob** será un objeto del tipo pasado en **T**. Por ejemplo, si el tipo **String** es pasado en **T**, entonces en dicha instancia, **ob** será de tipo **String**.

Ahora considérese el constructor de **Gen**:

```
Gen (T o) {
  ob = o;
}
```

Obsérvese que **ob** también es de tipo **T**. Esto significa que el tipo de **o** está determinado por el tipo pasado en **T** cuando se crea un objeto de la clase **Gen**. Además debido a que tanto el parámetro **o** como la variable **ob** son de tipo **T**, ambos serán del mismo tipo cuando un objeto de la clase **Gen** sea creado.

El parámetro **T** también puede ser utilizado para especificar el tipo de dato a ser devuelto por un método, como en el caso del método **getob( )** mostrado a continuación:

```
T getob( ) {
  return ob;
}
```

Debido a que **ob** también es de tipo **T**, su tipo es compatible con el tipo de retorno especificado por **getob( )**.

El método **showType( )** muestra el tipo de **T** mediante la llamada al método **getName( )** sobre el objeto de tipo **Class** devuelto por la llamada a **getClass( )** sobre **ob**. El método **getClass( )** está definido por la clase **Object** y es por ello un miembro de todos los tipos de clases. Este método devuelve un objeto **Class** que corresponde al tipo de la clase del objeto sobre el cuál es llamado. **Class** define el método **getName( )**, el cual regresa una cadena representativa del nombre de la clase.

La clase **GenDemo** demuestra como utilizar la clase genérica Gen. Primeramente crea una versión de Gen para enteros como se muestra aquí:

```
Gen<Integer> iOb;
```

Observe detalladamente esta declaración. Primero, note que el tipo **Integer** está especificado dentro de paréntesis angulares después del nombre **Gen**. En este caso, **Integer** es el *argumento de tipo* que es pasado al parámetro de tipo **T** en la clase **Gen**. Esta línea crea una versión de **Gen** en la cual todas las referencias a **T** son trasladadas en referencias a **Integer**, y el tipo de retorno para el método **getob( )** también es **Integer**.

Antes de continuar, es necesario aclarar que el compilador de Java no crea diferentes versiones de **Gen**, o de ninguna otra clase genérica. Aunque es útil pensar en esos términos, no es en realidad lo que pasa. En su lugar, el compilador elimina toda la información de los tipos parametrizados, sustituyéndolas por las conversiones de tipos necesarias, para hacer que el código se *comporte* como si la versión especificada de la clase fuera creada. Esto es, en el caso de nuestro ejemplo, existe solamente una versión de la clase **Gen**. El proceso de eliminar la información de los tipos parametrizados es llamada"*cancelación*", y hablaremos de ese tema más adelante en este capítulo.

La siguiente línea asigna a **iOb** una referencia a una instancia de una versión de la clase **Gen** que trabaja con elementos de tipo **Integer**:

```
iOb = new Gen<Integer> (88);
```

Nótese que cuando se llama al constructor de **Gen**, el argumento de especificación del tipo, **Integer**, también es especificado. Esto es necesario porque el tipo de objeto (en este caso **iOb**) al cual la referencia está siendo asignada es de tipo **Gen<Integer>**. Por eso, la referencia regresada por el operador **new** debe ser también de tipo **Gen<Integer>**. Si no se escribe de esta forma, se produce como resultado un error de compilación. Por ejemplo, la siguiente asignación causará un error de compilación:

```
iOb = new Gen<Double> (88.0); // Error
```

Debido a que **iOb** es de tipo **Gen<Integer>** no puede ser utilizado como referencia de un objeto de tipo **Gen<Double>**. Esta revisión de tipos es uno de los beneficios más importantes de los tipos parametrizados, porque asegura conversiones de tipos seguras.

Como los comentarios en el programa lo indican, la asignación

```
iOb = new Gen<Integer> (88);
```

hace uso del autoboxing para encapsular el valor 88, el cual es de tipo **int**, dentro de un **Integer**. Esto funciona debido a que **Gen<Integer>** crea un constructor que toma un argumento **Integer**. Dado que se espera un **Integer**, Java automáticamente aplica autoboxing para crear un Integer con el valor 88. Claro está que la asignación podría también haber sido escrita explícitamente, como se muestra a continuación:

```
iOb = new Gen<Integer> (new Integer(88));
```

Sin embargo, no habría ningún beneficio al utilizar esta versión.

El programa muestra el tipo de **ob** dentro de **iOb**, el cuál es **Integer**. A continuación, el programa obtiene el valor de **ob** con la siguiente línea:

```
int v = iOb.getob( );
```

Debido a que el tipo de regreso de **getob( )** es **T**, el cual fue reemplazado por **Integer** cuando **iOb** fue declarado, el tipo de regreso de **getob( )** es también **Integer**. Gracias al auto-unboxing, **Integer** se convierte en **int** cuando es asignado a **v** (la cual es de tipo **int**). Así, no hay necesidad

de convertir el tipo de regreso de **getob( )** a **Integer**. Claro está que no es necesario el uso de la característica auto-unboxing, la línea anterior podría haber sido escrita también como se muestra a continuación:

```
int v = iOb.getob( ).intValue( );
```

Sin embargo, la característica de auto-unboxing vuelve al código más compacto.

A continuación **GenDemo** declara un objeto de tipo **Gen<String>**:

```
Gen<String> strOb = new Gen <String> ("Prueba de tipos parametrizados");
```

Debido a que el argumento de tipo es **String**, **String** sustituye a **T** dentro de **Gen**. Esto crea (conceptualmente) una versión de **Gen** con **String**, como el resto de las líneas en el programa lo demuestran.

Los tipos parametrizados sólo trabajan con objetos

Cuando se declara una instancia con tipos parametrizados, el argumento de tipo que se pasa al parámetro de tipo debe ser una clase. No se pueden utilizar tipos primitivos, como **int** o **char**. Por ejemplo, con **Gen**, es posible pasar cualquier clase como valor para **T**, pero no se puede utilizar un tipo primitivo como valor de **T**. Por consiguiente, la siguiente línea es incorrecta:

```
Gen<int> strOb = new Gen<int>(53); //Error, no se pueden utilizar
tipos primitivos
```

Claro que la restricción de uso de los tipos primitivos no es una restricción seria porque se puede utilizar un envoltorio (como en el ejemplo anterior) para encapsular un tipo primitivo. Más aún, los mecanismos de autoboxing y auto-unboxing de Java hacen transparente el uso de la envoltura de tipos.

Los tipos parametrizados se diferencian por el tipo de sus argumentos

Un punto clave de entender acerca de los tipos parametrizados es que una referencia de una versión específica de un tipo parametrizado no es un tipo compatible con otra versión del mismo tipo parametrizado. Por ejemplo en el programa anterior, la siguiente línea de código es un error y no compilará:

```
iOb = strOb;  // error
```

Aunque tanto **iOb** como **strOb** son de tipo **Gen<T>**, son referencias a tipos diferentes debido a que sus parámetros de tipos son diferentes. Esto es parte del proceso en que los tipos parametrizados agregan seguridad y previenen errores.

Los tipos parametrizados son una mejora a la seguridad

En este punto, nos deberíamos estar preguntando lo siguiente: dado que las mismas funcionalidades proporcionadas por la clase genérica **Gen** puede ser lograda sin tipos parametrizados, simplemente especificando **Object** como el tipo de datos y empleando las conversiones de tipo correctas, ¿cuál es el beneficio de utilizar tipos parametrizados en la clase **Gen**? La respuesta es que los tipos parametrizados automáticamente garantizan seguridad en el manejo de tipos en todas las operaciones en las que se involucre **Gen**. En el proceso, los tipos

parametrizados eliminan la necesidad de codificar manualmente las operaciones de conversión de tipos y de comprobación de tipos.

Para entender los beneficios de los tipos parametrizados, consideremos el siguiente programa que crea un equivalente sin tipos parametrizados de **Gen**:

```
// La funcionalidad de la clase NoGen es equivalente a la de la clase Gen
// pero no utiliza tipos parametrizados
class NoGen {
  Object ob; // ob es ahora de tipo Object

  // Pasa al constructor una referencia a
  // un object de tipo Object
  NoGen(Object o) {
    ob = o;
  }

  // Devuelve un valor de tipo Object.
  Object getob( ) {
    return ob;
  }

  // Muestra el tipo de ob.
  void showType( ) {
    System.out.println("El tipo de ob es " +
                       ob.getClass( ).getName( ));
  }
}

// Utilizando la clase no genérica
class NoGenDemo {
  public static void main (String args[]) {
    NoGen iOb;

    // Crea un objeto NoGen y almacena
    // un valor de tipo Integer en él. El autoboxing ocurre igual que antes.
    iOb = new NoGen(88);

    // Muestra el tipo de dato utilizado por iOb.
    iOb.showType ( );

    // Obtiene el valor de iOb.
    // En este momento, es necesario hacer conversión de tipos
    int v = (Integer) iOb.getob( );
    System.out.println("valor: " + v);

    System.out.println( ) ;

    // Crea otro objeto NoGen y
    // almacena un String en él
    NoGen strOb = new NoGen("Prueba con tipos no parametrizados");

    // Muestra el tipo de datos usados por strOb.
    strOb.showType( ) ;

    // Obtiene el valor de strOb.
    // De nuevo, note que un conversión de tipos es necesaria.
```

```
    String str = (String) strOb.getob( );
    System.out.println("valor: " + str);

    // Este programa compila, pero es conceptualmente incorrecto
    iOb = strOb;.
    v = (Integer) iOb.getob( ); // error en tiempo de ejecución
  }
}
```

Existen muchas cosas interesantes en esta versión. Primero, note que **NoGen** reemplaza todas las ocurrencias de **T** con **Object**. Esto hace a **NoGen** capaz de almacenar cualquier tipo de objetos, tal como en la versión de los tipos parametrizados. Sin embargo, también evita que el compilador de Java tenga conocimiento real sobre el tipo de dato almacenado en **NoGen**, lo cual es malo por dos razones. Primero, deben emplearse conversiones explícitas de tipos para recuperar los datos almacenados. Segundo, no podrán ser detectados muchos errores de incompatibilidad de tipos sino hasta que el programa sea ejecutado. Veamos más de cerca cada problema.

Observemos la siguiente línea:

```
int v = (Integer) iOb.getob( );
```

Debido a que el tipo de retorno de **getob( )** es **Object**, la conversión a **Integer** es necesaria para que al valor **Integer** se le aplique auto-unboxing y sea almacenado en **v**. Si se elimina la conversión de tipos, el programa no compilará. Con la versión que utiliza tipos parametrizados, la conversión fue implícita. En la versión que no utiliza tipos parametrizados, la conversión debe ser explícita. Esto no sólo es incómodo, también es una fuente potencial de errores.

Ahora, considere la siguiente secuencia de instrucciones cerca del final del programa:

```
// Esto compila, pero es conceptualmente erróneo
iOb = strOb;
v = (Integer) iOb.getob( ); // error en tiempo de ejecución
```

Aquí, **strOb** es asignado a **iOb**. Sin embargo, **strOb** se refiere a un objeto que contiene una cadena, no a un entero. Esta asignación es sintácticamente válida porque todas las referencias a **NoGen** son iguales, y cualquier referencia a **NoGen** puede referirse a cualquier otro objeto **NoGen**. Sin embargo, la sentencia es semánticamente incorrecta. En estas líneas, el tipo de retorno de **getob( )** es convertido a **Integer**, y luego se hace un intento de asignar ese valor a **v**. El problema es que **iOb** ahora se refiere a un objeto que almacena un **String** y no un **Integer**. Desafortunadamente, sin el uso de tipos parametrizados el compilador de Java no tiene forma de saberlo. Por tanto, ocurre una excepción en tiempo de ejecución cuando se intenta realizar la conversión a **Integer**. Como ya sabrá, es extremadamente malo que el código tenga excepciones en tiempo de ejecución.

Las líneas anteriores no se presentan cuando se utilizan tipos parametrizados. Si esa secuencia fuera escrita en la versión del programa que utiliza tipos parametrizados, el compilador detectaría el problema y reportaría un error, de esta forma se previenen serios defectos, que a la postre podrían causar excepciones en tiempo de ejecución. La habilidad de crear código con tipos seguros en el cual los errores de incompatibilidad de tipos son eliminados en tiempo de compilación, es la ventaja clave de los tipos parametrizados. Aunque utilizar referencias de tipo **Object** para crear código con "tipos genéricos" siempre ha sido posible, el código no es seguro y su mal uso podría resultar en excepciones en tiempo de ejecución. Los tipos parametrizados previenen dichas excepciones. En esencia, a través de los

tipos parametrizados, los que eran errores en tiempo de ejecución se han convertido en errores en tiempo de compilación. Esta es la principal ventaja.

Una clase con tipos parametrizados con dos tipos como parámetro

Se puede declarar más de un parámetro de tipo en una clase genérica. Para especificar dos o más parámetros de tipo, simplemente se utiliza una lista separada con comas. Por ejemplo, la clase **DosGen** es una variación de la clase **Gen** con dos tipos parametrizados:

```
// Una clase simple con tipos parametrizados
// con dos parámetros de tipo: T y V.
class DosGen<T, V> {
  T ob1;
  V ob2;

  // Pasa al constructor como referencia
  // un objeto de tipo T y un objeto de tipo V.
  DosGen(T o1, V o2) {
    ob1 = o1;
    ob2 = o2;
  }

  // Muestra los tipos de T y V.
  void showTypes( ) {
    System.out.println("Tipo de T es" +
                       obl.getClass( ).getName( ));
    System.out.println("Tipo de V es" +
                       ob2.getClass( ).getName( ));

  T getobl( ) {
    return obl;
  }

  V getob2( ) {
    return ob2;
  }
}

// Ejemplo de uso de DosGen.
class SimpGen {
  public static void main(String args[]) {

    DosGen<Integer, String> tgObj =
      new DosGen<Integer, String> (88, "Tipos Parametrizados");

    // Muestra los tipos
    tgObj.showTypes( );

    // Obtiene y muestra los valores.
    int v = tgObj.getobl( );
    System.out.println ( "valor: " + v);

    String str = tgObj.getob2( );
    System.out.println("valor: " + str);
  }
}
```

La salida de este programa se muestra a continuación:

```
Tipo de T es java.lang.Integer
Tipo de V es java.lang.String
valor: 88
valor: Tipos Parametrizados
```

Nótese como se declara la clase **DosGen**:

```
class DosGen<T, V> {
```

Se especifican dos parámetros de tipo: **T** y **V**, separados por una coma. Debido a que tienen dos parámetros de tipo, dos argumentos de tipo deben ser pasados a la clase **DosGen** cuando se crea un objeto, como se muestra a continuación

```
DosGen <Integer, String> tgObj =
  new DosGen <Integer, String> (88, "Tipos Parametrizados");
```

En este caso, **Integer** es sustituido por **T**, y **String** sustituido por **V**.

Aunque los dos argumentos de tipo son diferentes en este ejemplo, es posible que ambos tipos sean iguales. Por ejemplo, la siguiente línea de código es válida:

```
DosGen <String, String> x = DosGen<String, String> ("A", "B");
```

En este caso, ambos **T** y **V** serían de tipo **String**. Claro que, si los argumentos de tipo fueran siempre los mismos, entonces tener dos parámetros de tipo sería innecesario.

La forma general de una clase con tipos parametrizados

La sintaxis de los tipos parametrizados mostrada en los ejemplos anteriores puede ser generalizada. Aquí está la sintaxis para declarar una clase con tipos parametrizados:

class *nombre-de-la-clase<lista de argumentos de tipo>*{ //...

Aquí está la sintaxis para la declaración a una referencia de una clase genérica:

nombre-de-la-clase <lista de argumentos de tipo > nombre-de-la-variable =
new nombre-de-la-clase <lista de argumentos tipo> (Lista de constantes);

Tipos delimitados

En los ejemplos anteriores, los parámetros tipo podrían ser reemplazados por cualquier clase. Esto es bueno para muchos propósitos, pero algunas veces es útil limitar los tipos que pueden ser pasados a un parámetro de tipo. Por ejemplo, asumiendo que se quiera crear una clase genérica que contenga un método que regrese el promedio de un arreglo de números. Más aún, se quiere utilizar la clase para obtener el promedio de un arreglo de cualquier tipo de números, incluyendo enteros, flotantes y reales. Esto es, se desea especificar el tipo de los números de forma general, utilizando un tipo parametrizado. Para crear tal clase, se podría intentar algo como lo que sigue:

```
// La clase Stats intenta (sin éxito)
// crear una clase con tipos parametrizados que puede calcular
// el promedio de un arreglo de números de
// cualquier tipo dado
```

```
//
// La clase contiene un error.
class Stats<T> {
  T[] nums;  // nums es un arreglo de tipo T

  // Pasa al constructor una referencia a
  // un arreglo de tipo T.
  Stats (T [] o) {
    nums = o;
  }

  // Devuelve tipo double en todos los casos
  double average( ) {
    double sum = 0.0;
    for(int i=0; i < nums.length; i++)
      sum += nums[i].doubleValue( ); //Error

    return sum / nums.length;
  }
}
```

En la clase **Stats**, el método **average( )** intenta obtener la versión de tipo **double** de cada número en el arreglo **nums** llamando al método **doubleValue( )**. Debido a que todas las clases numéricas, tales como **Integer** y **Double**, son subclases de **Number**, y **Number** define al método **doubleValue( )**, este método está disponible para todas las clases numéricas. El problema es que el compilador no tiene forma de saber que estamos intentando crear sólo objetos numéricos con la clase **Stats**. Por ello, cuando se intenta compilar **Stats**, se produce un error que indica que el método **doubleValue( )** no es conocido. Para resolver este problema, se necesita de alguna forma indicarle al compilador que la intención es pasar sólo tipos numéricos a **T**. Además se requiere alguna forma para asegurar que sólo sean pasados a **T** tipos numéricos.

Para manejar tales situaciones, Java provee *tipos delimitados*. Cuando se especifica un parámetro de tipo, se puede crear un delimitador superior que declara la superclase de la cual todos los argumentos de tipo deben estar derivados. Esto se logra utilizando una cláusula **extends** al especificar el parámetro de tipo, como se muestra a continuación:

<*T* extends *superclase*>

Esto especifica que *T* sólo puede ser reemplazado por *superclase* o subclases de *superclase*. De este modo, la superclase define un límite superior inclusivo.

Se puede utilizar un límite superior para solucionar el problema de la clase **Stats** mostrada anteriormente especificando **Number** como delimitador superior, como se muestra a continuación:

```
// En esta versión de Stats, el argumento de tipo para
// T debe ser Number, o una clase derivada
// de Number.
class Stats<T extends Number> {
  T[] nums; // arreglo de elementos de tipo Number o subclases de Number

  // Pasa al constructor una referencia a
  // un arreglo de valores de tipo Number o subclases de Number
  Stats (T [] o) {
    nums = o;
  }
```

```
  // Devuelve un valor de tipo double en todos los casos.
  double average( ) {
    double sum = 0.0;

    for(int i=0; i < nums.length; i++)
      sum += nums[i].doubleValue( );

    return sum / nums.length;
  }
}

// Muestra el uso de la clase Stats.
class BoundsDemo {
  public static void main(String args[]) {

    Integer inums[] = { 1, 2, 3, 4, 5 };
    Stats<Integer> iob = new Stats<Integer> (inums);
    double v = iob.average( );
    System.out.println("El promedio es" + v);

    Double dnums[] = { 1.1, 2.2, 3.3, 4.4, 5.5 };
    Stats<Double> dob = new Stats<Double>(dnums);
    double w = dob.average( );
    System.out.println("Promedio es:" + w);

    //  Esto no compilará porque String no es una
    //  subclase de Number.
//      String strs[] = { "1", "2", "3", "4", "5" };
//      Stats<String> strob = new Stats<String>(strs);

//      double x = strob.average( );
//      System.out.println("El promedio es" + v);

  }
}
```

La salida se muestra aquí:

```
Promedio es 3.0
Promedio es 3.3
```

Note como **Stats** ahora se declara como:

```
class Stats <T extends Number> {
```

Dado que el tipo **T** está ahora es delimitado por **Number**, el compilador de Java sabe que todos los objetos de tipo **T** puede llamar a **doubleValue( )** porque es un método declarado en la clase **Number**. Esto es, por sí mismo, una gran ventaja. Sin embargo, como un bono adicional, el delimitador de **T** también evita que se pase como parámetro a **Stats** un tipo no numérico. Por ejemplo, si se intenta eliminar el comentario de las líneas al final del programa, y recompilar el programa, se recibirá un error de compilación porque **String** no es una subclase de **Number**.

Además de utilizar una clase como tipo delimitado, se puede también utilizar una interfaz. De hecho, se pueden especificar múltiples interfaces como delimitadores. Más aún, un tipo delimitado puede incluir tanto una clase como una o más interfaces. En este caso, la clase debe ser especificada primero. Cuando un tipo delimitado incluye una interfaz, sólo son considerados

correctos los argumentos de tipo que hayan implementado la interfaz. Cuando se especifica un tipo delimitado que tiene una clase y una interfaz, o múltiples interfaces, se utiliza el operador **&** para conectarlas. Por ejemplo:

```
class Gen <T extends MiClase & MiInterfaz> {//...
```

Donde, **T** está delimitado por una clase llamada **MiClase** y una interfaz llamada **MiInterfaz**. De este modo, cualquier argumento de tipo pasado a **T** deberá ser una subclase de **MiClase** e implementar **MiInterfaz**.

Utilizando argumentos comodines

Son tan útiles como la seguridad de tipos, y algunas veces pueden generar construcciones perfectamente aceptables. Por ejemplo, considerando la clase **Stats** mostrada al final de la sección anterior, suponga que se desea agregar un método llamado **sameAvg( )** que determine si dos objetos **Stats** contienen arreglos que producen el mismo promedio, sin importar que tipos de datos numéricos contiene cada objeto. Por ejemplo, si un objeto contiene los valores de tipo **double** 1.0, 2.0 y 3.0, y el otro objeto contiene valores enteros 2, 1 y 3, entonces el promedio sería el mismo. Una forma de implementar **sameAvg( )** es pasarle un argumento de tipo **Stats**, y luego comparar el promedio del argumento contra el promedio del objeto que se invoca, devolviendo verdadero solo si el promedio es el mismo. Por ejemplo, si se quiere llamar al método **sameAvg( )**, como se muestra a continuación:

```
Integer inums[] = {1, 2, 3, 4, 5};
Double dnums[] = {1.1, 2.2, 3.3, 4.4, 5.5}

Stats <Integer> iob = new Stats <Integer>(inums);
Stats <Double> dob = new Stats <Double>(dnums);

if (ib.sameAvg(dob))
  System.out.println("El promedio es el mismo");
else
  System.out.println("El promedio es diferente");
```

Inicialmente, crear al método **sameAvg( )** parece ser un problema fácil. Debido a que **Stats** es una clase con tipos parametrizados y su método **average( )** puede funcionar con cualquier tipo de objeto **Stats**, parecería que crear al método **sameAvg( )** es simple. Desafortunadamente, el problema comienza tan pronto como se intenta declarar un parámetro de tipo **Stats**. Debido a que **Stats** es un tipo parametrizado, y la pregunta es ¿qué se especifica en el parámetro de tipo de **Stats** cuando se declara un parámetro de ese tipo?

Al principio se podría pensar en una solución como la siguiente, en la cual **T** se usa como tipo de parámetro:

```
// Esto no va a funcionar
// Determina si dos promedios son iguales.
boolean sameAvg(Stats<T> ob) {
  if (average( ) = = ob.average( ))
    return true;

  return false;
}
```

El problema con este intento es que funcionará solamente con otro objeto **Stats** cuyo tipo sea el mismo que el objeto que invoca. Por ejemplo, si el objeto que invoca es de tipo **Stats<Integer>**, entonces el parámetro **ob** debe también ser tipo **Stats<Integer>**. El método no puede ser utilizado para comparar el promedio de un objeto de tipo **Stats<Double>** con el promedio de un objeto de tipo **Stats<Short>**. Por consiguiente, esta estrategia no funcionará, excepto en un contexto muy limitado y no producirá una solución general.

Para crear un método **sameAvg( )** genérico, se debe utilizar otra característica de los tipos parametrizados de Java: los argumentos *comodines*. Los argumentos comodines son especificados por el símbolo **?**, que representa a un tipo desconocido. A continuación se muestra una forma de escribir el método **sameAvg( )** utilizando un comodín:

```
// Determina si dos promedios son iguales
// Note el uso de comodines.
boolean sameAvg(Stats<?> ob) {
  if (average( ) == ob.average( ))
    return true;

  return false;
}
```

Aquí **Stats<?>** se iguala con cualquier objeto **Stats**, lo cual permite comparar cualquier par de objetos **Stats**. El siguiente programa lo demuestra:

```
// Uso de comodines
class Stats<T extends Number> {
  T[] nums; // arreglo de valores Number o de alguna subclase de Number

  // Pasa al constructor una referencia a
  // un arreglo de tipo Number o subclase de Number
  Stats (T [] o) {
    nums = o;
  }

  // Devuelve un valor de tipo double en todos los casos.
  double average( ) {
    double sum = 0.0;

    for(int i=0; i < nums.length; i++)
      sum += nums[i].doubleValue( );

    return sum / nums.length;
  }

  // Determina si dos promedios son los mismos
  // Note el uso de comodines
  boolean SameAvg(Stats<?> ob) {
    if(average( ) = = ob.average( ))
      return true;

    return false;
  }
}
```

```
// Demuestra el uso de comodines
class ComodinDemo {
  public static void main(String args[]) {
    Integer inums[] = { 1, 2, 3, 4, 5 };
    Stats<Integer> iob = new Stats<Integer>(inums);
    double v = iob.average( );
    System.out.println("El promedio de iob es" + v);

    Double dnums [] = {1.1, 2.2, 3.3, 4.4, 5.5};
    Stats<Double> dob = new Stats<Double>(dnums);
    double w = dob.average( );
    System.out.println("El promedio de dob es" + w);

    Float fnums [] = { 1.0F, 2.0F, 3.0F, 4.0F, 5.0F };
    Stats<Float> fob = new Stats<Float> (fnums);
    double x = fob.average( );
    System.out.println("El promedio de fob es " + x) ;

    // Revisa cuál de los arreglos tiene el mismo promedio
    System.out.print ("El promedio de iob y dob");
    if(iob.sameAvg(dob))
      System.out.println("son iguales.");
    else
      System.out.println ("son diferentes ");

    System.out.print("El promedio de iob y fob ");
    if(iob.sameAvg(fob))
      System.out.println("son iguales.");
    else
      System.out.println("son diferentes.") ;
  }
}
```

La salida se muestra a continuación:

```
El promedio de iob es 3.0
El promedio de dob es 3.3
El promedio de fob es 3.0
Promedio de iob y dob son diferentes
Promedio de iob y fob son iguales.
```

Un último punto: Es importante entender que los comodines no afectan el tipo de objetos **Stats** que pueden ser creados. Esto está controlado por la cláusula **extends** en la declaración **Stats**. El comodín simplemente se iguala con cualquier objeto *válido* **Stats**.

Comodines delimitados

Los argumentos comodines pueden ser delimitados de la misma forma que un tipo parametrizado. Un comodín delimitado es especialmente importante cuando se está creando un tipo parametrizado que operará sobre una jerarquía de clases. Para entender porqué, veamos un ejemplo. Considere la siguiente jerarquía de clases que encapsulan coordenadas:

```
// Coordenadas bidimensionales.
class DosD {
  int x, y;

  DosD(int a, int b) {
    x = a;
    y = b;
  }
}

// Coordenadas tridimensionales.
class TresD extends DosD {
  int z;

  TresD(int a, int b, int c) {
    super (a, b);
    z = c;
  }
}

// Coordenadas en cuarta dimensión.
class CuatroD extends TresD {
  int t;

  CuatroD(int a, int b, int c, int d) {
    super (a, b, c);
    t = d;
  }
}
```

En la parte superior de la jerarquía está **DosD**, esta clase encapsula coordenadas bidimensionales XY. **DosD** es heredado por **TresD**, la cual agrega una tercera dimensión, creando coordenadas XYZ. **TresD** es heredado por **CuatroD**, la cual agrega una cuarta dimensión (tiempo), produciendo una coordenada de cuatro dimensiones.

A continuación se muestra una clase genérica llamada **Coords**, la cual almacena un arreglo de coordenadas:

```
//Esta clase contiene un arreglo de objetos coordenados
class Coords <T extends DosD> {
  T[] cords;

  Coords(T[] o) {cords = o;}
}
```

Note que **Coords** especifica un parámetro de tipo limitado por **DosD**. Esto significa que cualquier arreglo almacenado en un objeto **Coords** contendrá objetos de tipo **DosD** o una de sus subclases.

Ahora, asumiendo que se desea escribir un método que muestre las coordenadas X y Y para cada elemento en el arreglo **coords** del objeto de tipo **Coords**. Dado que todos los tipos de objetos **Coords** tienen al menos dos coordenadas (X y Y), es fácil de hacer esto utilizando un comodín, como se muestra a continuación:

```
static void muestraXY(Coords <?> c) {
  System.out.println("Coordenadas X Y: ");
  for (int i=0; i < c.coords.length; i++)
```

```
    System.out.println(c.coords[i].x + " " +
                       c.coords[i].y);
  System.out.println( );
}
```

Debido a que **Coords** es un tipo parametrizado limitado que especifica a **DosD** como su límite superior, todos los objetos que pueden ser utilizados para crear un objeto **Coords** serán arreglos de tipo **DosD**, o de clases derivadas de **DosD**. Así, el método **muestraXY( )** puede desplegar el contenido de cualquier objeto **Coords**.

Sin embargo, ¿qué pasa si se desea crear un método que muestre las coordenadas X, Y y Z de un objeto **TresD** o **CuatroD**? El problema es que no todos los objetos **Coords** tendrán tres coordenadas, un objeto **Coords<DosD>** sólo tendrá coordenadas X y Y. Por lo tanto, ¿cómo se escribiría un método que muestre las coordenadas X, Y y Z para un objeto **Coords<TresD>** y **Coords<CuatroD>**, impidiendo que el método sea utilizado por un objeto **Coords<DosD>**? La respuesta es el *argumento comodín delimitado*.

Un comodín delimitado especifica un límite superior o un límite inferior para el tipo de argumento. Esto permite restringir el tipo de objetos sobre el cual un método operará. El comodín delimitado más común es el límite superior, el cual se crea utilizando la cláusula **extends** de una forma muy similar a la que se usa para crear un tipo delimitado.

Utilizando un comodín delimitado, es fácil crear un método que muestre las coordenadas X, Y y Z de un objeto **Coords**, si el objeto en cuestión tiene tres coordenadas. Por ejemplo, el siguiente método **muestraXYZ( )** despliega las coordenadas X, Y y Z de los elementos almacenados en el objeto **Coords**, si dichos elementos son de tipo **TresD** (o son derivados de **TresD**):

```
static void muestraXYZ(Coords<? extends TresD> c) {
  System.out.println("Coordenadas X Y Z: ");
  for (int i=0; i < c.coords.lenght; i++)
    System.out.println(c.coords[i].x + " " +
                       c.coords[i].y + " " +
                       c.coords[i].z);
  System.out.println( );
}
```

Note que ha sido agregada una cláusula **extends** en la declaración del comodín delimitado con el parámetro **c**. Esto declara que el símbolo **?** puede corresponder con cualquier tipo siempre y cuando sea **TresD**, o una clase derivada de **TresD**. Así la cláusula **extends** establece un límite superior que el símbolo **?** debe satisfacer. Debido a esta delimitación, el método **muestraXYZ( )** puede ser llamado con referencias a objetos de tipo **Coords<TresD>** o **Coords<CuatroD>**, pero no con referencia a tipos **Coords<DosD>**.

Intentar llamar a **muestraXYZ( )** con una referencia a **Coords<DosD>** causará un error en tiempo de compilación, de esa manera se garantiza seguridad en el manejo de tipos.

A continuación se presenta un programa que muestra en acción a los argumentos con comodines delimitados:

```
// Argumentos con comodines delimitados

// Coordenadas bidimensionales
class DosD {
  int x, y;
```

```
  DosD (int a, int b) {
    x = a;
    y = b;
  }
}

// Coordenadas tridimensionales.
class TresD extends DosD {
  int z;

  TresD(int a, int b, int c) {
    super(a, b);
    z = c;
  }
}

// Coordenadas en cuarta dimensión.
class CuatroD extends TresD {
  int t;

  CuatroD(int a, int b, int c, int d) {
    super (a, b, c);
    t = d;
  }
}

// Esta clase contiene un arreglo de objetos para coordenadas.
class Coords<T extends DosD> {
  T[] coords;

  Coords(T[] o) { coords = o; }
}

// Demuestra el uso de comodines delimitados.
class ComodinDelimitado {
  static void showXY(Coords<?> c) {
    System.out.println("Coordenadas XY:");
    for(int i=0; i < c.coords.length; i++)
      System.out.println(c.coords[i).x + " " +
                         c.coords[i).y);
    System.out.println( );
  }

  static void showXYZ(Coords<? extends TresD> c) {
    System.out.println("Coordenadas XYZ: ");
    for(int i=0; i < c.coords.length; i++)
      System.out.println(c.coords[i].x + " " +
                         c.coords[i).y + " " +
                         c.coords [i].z);
    System.out.println( );
}

static void showAll(Coords<? extends CuatroD> c) {
  System.out.println("Coordenadas XYZT:");
  for(int i=0; i < c.coords.length; i++)
    System.out.println(c.coords[i).x + " " +
                       c.coords[i].y + " " +
```

```
                        c.coords[i].z + " " +
                        c.coords [i].t);
  System.out.println( ) ;
}

public static void main(String args[]) {
  DosD td[] = {
      new DosD (0, 0),
      new DosD(7, 9),
      new DosD (18, 4),
      new DosD(-1, -23)
    };

    Coords<DosD> tdlocs = new Coords<DosD>(td);

    System.out.println("Contenido de tdlocs.");
    showXY(tdlocs);   // bien, es un DosD
//    showXYZ(tdlocs);  // Error, no es un TresD
//    showAll(tdlocs);  // Error, no es un CuatroD

      // Ahora, creamos algunos objetos CuatroD
      CuatroD fd[] = {
      new CuatroD(1, 2, 3, 4),
      new CuatroD(6, 8, 14, 8),
      new CuatroD(22, 9, 4, 9),
      new CuatroD(3, -2, -23, 17)
    };

    Coords<CuatroD> fdlocs = new Coords<CuatroD> (fd) ;

    System.out.println("Contenido de fdlocs.");
    // Todos estos están correctos.
    showXY(fdlocs);
    showXYZ (fdlocs);
    showAll(fdlocs);
  }
}
```

La salida del programa se muestra a continuación:

```
Contenido de tdlocs.
Coordenadas XY:
0 0
7 9
18 4
-1 -23

Contenido de fdlocs.
Coordenadas XY:
1 2
6 8
22 9
3 -2

Coordenadas XYZ:
1 2 3
6 8 14
22 9 4
3 -2 -23
```

```
Coordenadas X Y Z T:
1 2 3 4
6 8 14 8
22 9 4 9
3 -2 -23 17
```

Note las siguientes líneas comentadas:

```
// showXYZ(tdlocs);  // Error, no es un TresD
// showAll(tdlocs);  // Error, no es un CuatroD
```

Debido a que **tdlocs** es un objeto **Coords(DosD)**, no puede ser utilizado para llamar a **showXYZ( )** o **showAll( )** debido a que el argumento de comodín delimitado lo impide. Para probarlo, intentemos remover los símbolos de comentario y luego compilar el programa; recibirá errores de compilación debido a la incompatibilidad de tipos.

En general, para establecer un límite superior para un comodín delimitado, se utiliza la siguiente expresión de comodín:

<? extends *superclase*>

donde *superclase* es el nombre de la clase que sirve como límite superior. Recuerde que ésta es una cláusula inclusiva porque la clase que define el límite superior también está considerada dentro del límite.

También se puede especificar un límite inferior para un comodín delimitado, agregando una cláusula **super** a la declaración del comodín delimitado. A continuación su forma general:

<? super *subclase*>

En este caso, sólo las clases que son superclases de *subclase* son argumentos aceptables. Ésta es una cláusula exclusiva, porque no se considera a la *subclase* como parte del límite aceptable.

Métodos con tipos parametrizados

Como se mostró en los ejemplos anteriores, los métodos dentro de una clase genérica pueden hacer uso de los parámetros de tipo de la clase y por consiguiente están automáticamente ligados al tipo del parámetro. Sin embargo, es posible declarar un método genérico que utilice uno o más parámetros de tipo propios. También es posible crear métodos genéricos contenidos en clases no genéricas.

Comencemos con un ejemplo. El siguiente programa declara una clase no genérica llamada **GenMethDemo** y un método estático genérico dentro de esa clase llamado **estaEn( )**. El método **estaEn( )** determina si un objeto es miembro de un arreglo. Este método puede ser usado con cualquier tipo de objetos y arreglos siempre y cuando el arreglo contenga objetos que sean compatibles con el tipo de los objetos que están siendo buscados.

```
// Ejemplo de método con tipos parametrizados
class GenMethDemo{

  //Determina si un objeto está en un arreglo
  static <T, V extends T> boolean estaEn(T x, V[] y){
```

```
    for (int i=0; i < y.length; i++)
      if (x.equals(y[i])) return true;

    return false;
  }

  public static void main(String args[]) {

    // Utiliza estaEn( ) sobre Integers.
    Integer nums[] = { 1, 2, 3, 4, 5 };

    if (estaEn(2, nums))
      System.out.println("2 está en nums");

    if (!estaEn(7, nums))
      System.out.println("7 no está en nums");

    System.out.println( ) ;

    // Usa estaEn( ) sobre Strings.
    String strs[] = { "uno", "dos", "tres" ,
                      "cuatro", "cinco" };

    if (estaEn("dos", strs))
      System.out.println{"dos está en strs");

    if (!estaEn("siete", strs))
      System.out.println{"siete no está en strs");

    // ¡ups, no compilará! Los tipos deben ser compatibles.
//  if (estaEn("dos", nums))
// System.out.println("dos está en strs");
  }
}
```

La salida del programa se muestra a continuación:

```
2 está en nums
7 no está en nums

dos está en strs
siete no está en strs
```

Examinemos **estaEn( )** más de cerca. Primero, note como se declara el método en la siguiente línea:

```
static <T, V extends T> boolean estaEn(T x, V[] y) {
```

Los parámetros de tipo están declarados *antes* del tipo de retorno del método. Segundo, note que el tipo **V** está limitado por **T**. Así, **V** debe ser el mismo tipo **T**, o una subclase de **T**. Esta relación exige que **estaEn( )** puede ser llamado sólo con argumentos que son compatibles. También note que **estaEn( )** es estático, habilitándolo para ser llamado independientemente de cualquier objeto. Sin embargo, debe tenerse en cuenta que los métodos genéricos pueden ser tanto estáticos como no estáticos. No hay restricción en este sentido.

Ahora, note como **estaEn( )** es llamado dentro de **main( )** utilizando una sintaxis tradicional, sin la necesidad de especificar algún argumento de tipo. Esto es debido a que los

argumentos de tipos son automáticamente percibidos, y los tipos de **T** y **V** son ajustados como corresponde. Por ejemplo, en la primera llamada:

```
if (estaEn(2, nums))
```

el tipo del primer argumento es **Integer** (debido al autoboxing), lo cuál causa que **Integer** sea sustituido por **T**. El tipo base del segundo argumento es también **Integer**, lo cual ocasiona que **Integer** también sea sustituido por **V**.

En la segunda llamada, el tipo **String** se utiliza como tipo para **T** y **V**. Observe el código comentado, mostrado a continuación:

```
//    if (estaEn("dos", nums))
//      System.out.println("dos está en strs");
```

Si se remueven los comentarios y después se intenta compilar el programa, se recibirá un mensaje de error. La razón es que el parámetro de tipo **V** está limitado por **T** con la cláusula **extends** en la declaración de **V**. Esto significa que **V** debe ser de tipo **T** o una subclase de **T**. En este caso, el primer argumento es de tipo **String**, haciendo a **T** de tipo **String**, pero el segundo argumento es de tipo **Integer**, el cual no es una subclase de **String**. Esto genera un error de tipos incompatibles en tiempo de compilación. Esta habilidad de forzar la seguridad en el manejo de tipos es una de las ventajas más importantes de los métodos con tipos parametrizados.

La sintaxis utilizada para crear **estaEn( )** puede ser generalizada. A continuación se muestra la sintaxis para métodos con tipos parametrizados.

<lista-param-tipo>tipo-retorno nombre-metodo(*lista-parametros*){//...

En todos los casos, *lista-param-tipo* es una lista de tipos de parámetros separados por comas. Note que para un método con tipos parametrizados, la lista de tipos de parámetros precede al tipo de valor devuelto por el método.

Constructores con tipos parametrizados

También es posible hacer constructores con tipos parametrizados, incluso si sus clases no lo son. Por ejemplo, considere el siguiente programa:

```
// Uso de constructores con tipos parametrizados.
class GenCons {
  private double val;

  <T extends Number> GenCons(T arg) {
    val = arg.doubleValue( );
  }

  void showval( ) {
    System.out.println ("valor: " + val);
  }
}

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

    GenCons test = new GenCons(l00);
    GenCons test2 = new GenCons(123.5F);
```

```
    test.showval( );
    test2.showval( );
  }
}
```

La salida se muestra aquí:

```
valor: 100.0
valor: 123.5
```

Dado que **GenCos( )** especifica un parámetro genérico, el cual debe ser una subclase de **Number**, **GenCos( )** puede ser llamado con cualquier tipo numérico, incluyendo **Integer**, **Float**, o **Double**. Por lo tanto, aunque **GenCos** no es una clase genérica, su constructor es genérico.

Interfaces con tipos parametrizados

Además de las clases y métodos con tipos parametrizados, se pueden también tener interfaces con tipos parametrizados. Las interfaces parametrizadas se especifican igual que las clases parametrizadas. Veamos un ejemplo, que crea una interfaz llamada **MinMax** que declara los métodos **min( )** y **max( )**, los cuales se espera regresen el valor mínimo y el valor máximo de un conjunto de objetos.

```
// Un ejemplo de interfaz con tipos parametrizados

// La interfaz MinMax
interface MinMax<T extends Comparable<T>> {
  T min( );
  T max( );
}

// Ahora, una implementación de MinMax
class MiClase<T extends Comparable<T>> implements MinMax<T> {
  T[] vals;

  MiClase(T[] o) { vals = o; }

  // Devuelve el valor mínimo en vals.
  public T min ( ) {
    T v = vals[0];

    for(int i=1; i < vals.length; i++)
      if(vals[i].compareTo(v) < 0) v = vals[i];

    return v;
  }

  // Devuelve el valor máximo en vals.
  public T max ( ) {
    T v = vals [0];

    for(int i=1; i < vals.length; i++)
      if (vals [i].compareTo(v) > 0) v = vals[i];

    return v;
  }
}
```

```
class GenIFDemo {
  public static void main(String args[]) {
    Integer inums[] = {3, 6, 2, 8, 6 };
    Character chs [] = {'b', 'r', 'p', 'w'};

    MiClase<Integer> iob = new MiClase<Integer> (inums);
    MiClase<Character> cob = new MiClase<Character>(chs);

    System.out.println("Valor máximo en inums: " + iob.max( ));
    System.out.println("Valor mínimo en inums: " + iob.min( ));

    System.out.println("Valor máximo en chs: " + cob.max( ));
    System.out.println("Valor mínimo en chs: " + cob.min{));
  }
}
```

La salida se muestra a continuación:

```
Valor máximo en inums: 8
Valor mínimo en inums: 2
Valor máximo en chs: w
Valor mínimo en chs: b
```

Aun cuando la mayoría de los aspectos de este programa deberían ser fáciles de entender, es necesario realizar un par de observaciones. Primero, note que **MinMax** está declarada como sigue:

```
interface MinMax<T extends Comparable<T>> {
```

En general, una interfaz con tipos parametrizados se declara de la misma forma que una clase con tipos parametrizados. En este caso, el tipo de parámetro es **T**, y su límite superior es **Comparable**, la cual es una interfaz definida por **java.lang**. Una clase que implementa a **Comparable** define objetos que pueden ser ordenados. De esta forma, usar a **Comparable** como límite superior asegura que **MinMax** sólo puede ser utilizada con objetos que son capaces de ser comparados (véase el Capítulo 16 para mayor información de **Comparable**). Nótese que **Comparable** es también una interfaz genérica (fue mejorada en JDK 5). **Comparable** tiene un parámetro de tipo que especifica el tipo de los objetos que se están comparando.

A continuación, **MiClase** implementa a **MinMax**. Nótese la declaración de **MiClase**, que se muestra aquí:

```
class MiClase<T extends Comparable<T> implements MinMax<T> {
```

Pongamos especial atención en la forma en que el parámetro de tipo, llamado **T**, está declarado por **MiClase** y luego es pasado a **MinMax**. Debido a que **MinMax** requiere un tipo que implemente de **Comparable**, la clase implementada (**MiClase** en este caso) debe especificar el mismo límite. Más aún, una vez que dicho límite ha sido establecido, no hay necesidad de especificarlo de nuevo en la cláusula de **implementación**. De hecho, estaría mal hacerlo. Por ejemplo, esta línea es incorrecta y no compilará:

```
//Esto está mal.
class MiClase <T extends Comparable <T>>
       implements MinMax<T extends Comparable<T>> {
```

Una vez que el tipo de parámetro ha sido establecido, simplemente se pasa a la interfaz sin mayor modificación.

En general, si una clase implementa de una interfaz con tipos parametrizados, entonces las clases también deben ser de tipos parametrizados, al menos extender de una, ya que requiere un parámetro de tipo para pasarlo a la interfaz. Por ejemplo, el siguiente intento de declarar **MiClase** es un error:

```
class MiClase implements MinMax<T> { // error
```

Dado que **MiClase** no declara un parámetro de tipo, no hay forma de pasar uno a **MinMax**. En este caso, el identificador **T** es simplemente desconocido, y el compilador reportará un error. Claro está, que si una clase implementa a la interfaz genérica proporcionando un *tipo especifico*, como la que se muestra a continuación:

```
class MiClase implements MinMax<Integer> { // correcto
```

entonces la implementación de la clase no necesita utilizar tipos parametrizados.

La interfaz con tipos parametrizados ofrece dos beneficios. Primero, puede ser implementada por diferentes tipos de datos. Segundo, permite colocar restricciones (esto es, límites) sobre el tipo de dato con los cuales la interfaz puede ser implementada. En el ejemplo de **MinMax**, sólo tipos que implementan de la interfaz **Comparable** pueden ser pasados a **T**.

Aquí está la sintaxis generalizada para una interfaz con tipos parametrizados:

interface *nombre-interfaz* <*tipo-param-lista*> {//…

Donde, *tipo-param-lista* debe ser una lista de parámetros de tipo separados por coma. Cuando una interfaz con tipos parametrizados es implementada, es necesario especificar los argumentos de tipo, como se muestra a continuación:

class *nombre-clase* <*tipo-param-lista*>
    implements *nombre-interfaz*<*tipo-arg-lista*> {

Compatibilidad entre el código de versiones anteriores y los tipos parametrizados

Dado que el soporte para tipos parametrizados es una adición reciente a Java, fue necesario proveer algún camino de transición del código viejo previo a los tipos parametrizados. Al momento de estar escribiendo este libro, existen aún millones y millones de líneas de código sin tipos parametrizados que se deben mantener funcionales y compatibles con código nuevo que utiliza tipos parametrizados. Los códigos previos a los tipos parametrizados deben ser capaces de funcionar con tipos parametrizados y el código con tipos parametrizados debe ser capaz de funcionar con código previo a los tipos parametrizados.

Para gestionar la transición hacia tipos parametrizados, Java permite a una clase con tipos parametrizados ser utilizada sin ningún argumento de tipo. Esto crea un *tipo en bruto* para la clase. Este tipo en bruto es compatible con los códigos anteriores, que no tiene conocimiento de los tipos parametrizados. El principal inconveniente de utilizar el tipo en bruto es que la seguridad en el manejo de tipos proporcionada por el uso de tipos parametrizados se pierde.

A continuación se muestra un ejemplo:

```
// Uso de un tipo en bruto
class Gen<T> {

  T ob;  // declara un objeto de tipo T

  // Pasa al constructor una referencia a
  // un objeto de tipo T.
  Gen(T o} {
    ob = o;
  }

  // Devuelve ob.
  T getob ( ) {
    return ob;
  }
}

// Uso del tipo en bruto.
class RawDemo {
  public static void main(String args[]) {

    //  Crea un objeto de tipo Gen para Integer.
    Gen<Integer> iOb = new Gen<Integer> (88);

    //  Crea un objeto Gen para String.
    Gen<String> strOb : new Gen<String> ("Prueba de tipos parametrizados"};

    //  Crea un objeto Gen con tipo en bruto y le asigna
    //  un valor Double
    Gen raw = new Gen(new Double(98.6));

    //  Es necesario hacer una conversión de tipos aquí,
        dado que el tipo es desconocido
    double d = (Double) raw.getob( );
    System.out.println ("valor: " + d);

    //  El uso de un tipo en bruto puede generar una excepción en tiempo
    //  de ejecución. Aquí tenemos algunos ejemplos.

    //  La siguiente conversión causa un error en tiempo de ejecución
//      int i = (Integer) raw.getob( ); // error en tiempo de ejecución

    //  Esta asignación pasa por alto la seguridad de tipos
    strOb = raw; // es correcto, pero potencialmente erróneo
//      String str = strOb.getob( ); // error en tiempo de ejecución

      // Esta asignación también pasa por alto la seguridad de tipos
      raw = iOb; // es correcto, pero potencialmente erróneo
//      d = (Double) raw.getob( ); // error en tiempo de ejecución
  }
}
```

Este programa contiene muchas cosas interesantes. Primero, un objeto de tipo **Gen** con tipo parametrizado en bruto se crea mediante la siguiente declaración:

```
Gen raw = new Gen(new Double(98.6));
```

Note que no hay argumentos de tipos especificados. En esencia, esto crea un objeto **Gen** cuyo tipo **T** se reemplaza por **Object**.

Un tipo en bruto no es seguro. De esta manera, una variable de un tipo en bruto puede ser asignada como referencia a cualquier tipo de objeto **Gen**. Al inverso también está permitido; una variable de un tipo específico **Gen** puede ser asignada como referencia a un objeto **Gen** de tipo en bruto. Sin embargo, ambas operaciones son potencialmente inseguras porque el mecanismo de revisión de tipos parametrizados es evadido.

Esta falta de seguridad se ilustra con las líneas comentadas al final del programa. Examinemos cada caso. Primero, considere la siguiente situación:

```
// int i = (Integer) raw.getob( ); // error en tiempo de ejecución
```

En esta sentencia, se obtiene el valor del atributo **ob** del objeto llamado **raw**, y su valor es convertido en un **Integer**. El problema es que el objeto **raw** contiene un valor **Double**, no un valor **Integer**. Sin embargo, esto no puede ser detectado en tiempo de compilación puesto que el tipo del objeto **raw** se desconoce. De esta forma, esta sentencia falla en tiempo de ejecución.

La siguiente secuencia asigna a **strOb** (referencia de tipo **Gen<String>**) una referencia a un objeto **Gen** de tipo bruto:

```
strOb = raw; // es correcto, pero potencialmente erróneo
//    String str = strOb.getob( ); // error en tiempo de ejecución
```

Esta sentencia, por sí misma, es sintácticamente correcta, pero cuestionable. Puesto que **strOb** es de tipo **Gen<String>**, se asume que contiene un **String**. Sin embargo, después de la asignación, el objeto referido por **strOb** contiene un **Double**. De esta manera, en tiempo de ejecución, cuando se intente asignar el contenido de **strOb** a **str**, el resultado será un error en tiempo de ejecución, porque **strOb** contiene un **Double**. Así, la asignación de un tipo en bruto por referencia a un tipo parametrizado pasa de lado el mecanismo de revisión de seguridad de tipos.

La siguiente secuencia invierte el caso anterior

```
   raw = iOb; // es correcto, pero potencialmente erróneo
//   d = (Double) raw.getob( ); // error en tiempo de ejecución
```

Aquí, un tipo parametrizado se asigna a una referencia de una variable de tipo en bruto. Aunque esta sentencia es sintácticamente correcta, puede dar problemas, como se ilustra en la segunda línea. En este caso, el objeto **raw** hace referencia a un objeto que contiene un objeto **Integer**, pero la conversión asume que contiene un **Double**. Este error no se puede prevenir en tiempo de compilación. Por el contrario, causa un error en tiempo de ejecución.

A causa del peligro potencial inherente a los tipos en brutos, **javac** muestra una *advertencia de tipos no comprobados* cuando un tipo en bruto es utilizado en una forma que podría poner en peligro la seguridad de tipos. En el programa anterior, las siguientes líneas provocan advertencias de tipos no comprobados:

```
Gen raw = new Gen(new Double(98.6));

strOb = raw; // es correcto, pero potencialmente erróneo
```

En la primera línea, la llamada al constructor **Gen** sin el argumento de tipo causa la advertencia. En la segunda línea, la asignación de una referencia de tipo en bruto a una variable de tipo parametrizado es lo que genera la advertencia.

Al principio, se podría pensar que la siguiente línea debería generar también una advertencia de tipo no comprobado, pero no lo hace:

```
raw = iOb; // es correcto, pero potencialmente erróneo
```

No se genera ninguna advertencia en tiempo de compilación porque la asignación no causa una pérdida de *seguridad* en el manejo de tipos más allá de la que ya ha ocurrido cuando el objeto llamado **raw** fue creado.

Un punto final: se debe limitar el uso de tipos en brutos a aquellos casos en los cuales se requiere mezclar código antiguo con código nuevo con tipos parametrizados. Los tipos en bruto son simplemente una característica de transición y no algo que deba ser utilizado en códigos nuevos.

Jerarquía de clases con tipos parametrizados

Las clases con tipos parametrizados pueden ser parte de una jerarquía de clases de la misma forma en la que lo son las clases sin tipos parametrizados. De esta forma, una clase con tipos parametrizados puede actuar como una superclase o ser una subclase. La diferencia clave entre las jerarquías de clases con tipos parametrizados y sin tipos parametrizados es que en una jerarquía con tipos parametrizados, cualquier argumento de tipo que sea requerido por una superclase con tipos parametrizados debe ser proporcionado a la jerarquía por todas las subclases. Esto es similar a la forma en que los argumentos del constructor se pasan en la jerarquía.

Superclases con tipos parametrizados

El siguiente es un ejemplo simple de una jerarquía que utiliza una superclase con tipos parametrizados:

```
// Una jerarquía de clases simples con tipos parametrizados
class Gen<T> {
  T ob;

  Gen(T o) {
    ob = o;
  }

  // Devuelve ob.
  T getob ( ) {
    return ob;
  }
}

// Una subclase de Gen.
class Gen2<T> extends Gen<T> {
  Gen2(T o) {
    super (o) ;
  }
}
```

En esta jerarquía, **Gen2** extiende la clase genérica **Gen**. Note que **Gen2** se declara con la siguiente línea:

```
class Gen2<T> extends Gen<T> {
```

El parámetro de tipo **T** es especificado por **Gen2** y también es pasado a **Gen** en la cláusula **extends**. Esto significa que cualquier tipo que se pasa a **Gen2** también se pasará a **Gen**. Por ejemplo, la siguiente declaración:

```
Gen2<Integer> num = new Gen2<Integer> (100);
```

pasa **Integer** como el parámetro de tipo para **Gen**. De esta forma, el atributo **ob** dentro de **Gen** que es parte también de **Gen2** será de tipo **Integer**.

Note también que **Gen2** no utiliza el parámetro de tipo **T** excepto para pasarlo a la superclase **Gen**. De esta manera, incluso si una subclase de una superclase genérica no requiere ser genérica, aún así debe especificar el parámetro de tipo requerido por su superclase genérica.

Claro que, una subclase es libre de agregar sus propios parámetros, si los requiere. Por ejemplo, aquí está una variación de la jerarquía anterior en la cual **Gen2** agrega sus propios tipos parametrizados:

```
// Una subclase puede agregar sus propios parámetros de tipo
class Gen<T> {
  T ob; // declara un objeto de tipo T

  // Pasa al constructor una referencia a
  // un objeto de tipo T.
  Gen(T o) {
    ob = o;
  }

  // Devuelve ob.
  T getob ( ) {
    return ob;
  }
}

// Una subclase de Gen que define un segundo
// tipo de parámetro, llamado V.
class Gen2<T, V> extends Gen<T> {
  V ob2;

  Gen2(T o, V o2) {
    super (o) ;
    ob2 = o2;
  }

  V getob2( ) {
    return ob2;
  }
}

// Crea un objeto de tipo Gen2.
class JerarquiaDemo {
  public static void main(String args[]) {
```

```
  // Crea un objeto Gen2 para String e Integer.
  Gen2<String, Integer> x =
    new Gen2<String, Integer>("El valor es: ", 99);

    System.out.print(x.getob( ));
    System.out.println(x.getob2( ));
  }
}
```

Observe la declaración de esta versión de **Gen2**, mostrada a continuación:

```
class Gen2<T, V> extends Gen<T> {
```

Donde **T** es el tipo que se pasa a **Gen**, y **V** es el tipo que se específica en **Gen2**. **V** se utiliza para declarar un objeto llamado **ob2**, y como tipo de retorno para el método **getob2( )**. En el método **main( )** se crea un objeto **Gen2** en el cual el parámetro de tipo **T** es **String**, y el parámetro de tipo **V** es **Integer**. El programa muestra el siguiente resultado:

```
El valor es: 99
```

Subclases con tipos parametrizados

Es perfectamente válido para una clase sin tipos parametrizados ser la superclase de una subclase con tipos parametrizados. Por ejemplo, considere el siguiente programa:

```
// Una clase sin tipos parametrizados puede ser una superclase
// de una subclase con tipos parametrizados

// Una clase sin tipos parametrizados
class NoGen {
  int num;

  NoGen(int i) {
    num = i;
  }

  int getnum( ) {
    return num;
  }
}

// Una subclase con tipos parametrizados
class Gen<T> extends NoGen {
  T ob;  // declara un objeto de tipo T

  // Pasa al constructor una referencia a
  // un objeto de tipo T.
  Gen(T o, int i) {
    super(i) ;
    ob = o;
  }

  // Devuelve ob.
  T getob( ) {
    return ob;
  }
}
```

```
// Crea un objeto de tipo Gen
class JerarquiaDemo2 {
  public static void main(String args[]) {

    // Crea un objeto Gen para String.
    Gen<String> w = new Gen<String> ("Hola", 47);

    System.out.print(w.getob( ) + " ");
    System.out.println(w.getnum( )) ;
  }
}
```

La salida del programa se muestra a continuación:

```
Hola 47
```

En el programa, note como **Gen** hereda de **NoGen** en la siguiente declaración:

```
class Gen<T> extends NoGen {
```

Dado que **NoGen** no utiliza tipos parametrizados, no se le especifica ningún argumento de tipo. Así, aunque **Gen** declara el parámetro de tipo **T**, éste no es requerido (ni puede ser utilizado) por **NoGen**. De esta forma, **NoGen** es heredado por **Gen** en la forma normal. No se aplica ninguna condición especial.

Comparación de tipos en tiempo de ejecución

Recordemos al operador **instanceof** descrito en el Capítulo 13. Como se explicó, **instanceof** determina si un objeto es una instancia de una clase. El operador devuelve verdadero si un objeto pertenece al tipo especificado o bien puede ser convertido en dicho tipo. El operador **instanceof** puede ser aplicado a objetos de clases con tipos parametrizados. La siguiente clase es un ejemplo de las implicaciones de una jerarquía de clases con tipos parametrizados en la compatibilidad de tipos:

```
//  Uso del operador instanceof con una jerarquía de clases
    con tipos parametrizados
class Gen<T> {
  T ob;

  Gen(T o) {
    ob = o;
  }

  // Devuelve ob.
  T getob ( ) {
    return ob;
  }
}

// Una subclase de Gen.
class Gen2<T> extends Gen<T> {
  Gen2 (T o) {
    super (o) ;
  }
}
```

```
// Implicaciones de los tipos parametrizados en jerarquía de clases
// con el operador instanceof
class JerarquiaDemo3 {
  public static void main(String args[]) {

    // Crea un objeto Gen para objetos Integer.
    Gen<Integer> iOb = new Gen<Integer> (88);

    // Crea un objeto Gen2 para objetos Integer.
    Gen2<Integer> iOb2 = new Gen2<Integer> (99);

    // Crea un objeto Gen2 para objetos String.
    Gen2<String> strOb2 = new Gen2<String> ("Prueba de tipos parametrizados");

    // Ve si iOb2 tiene alguna forma de Gen2.
    if (iOb2 instanceof Gen2<?>)
      System.out.println("iOb2 es instancia de Gen2");

    // Ve si iOb2 tiene alguna forma de Gen
    if (iOb2 instanceof Gen<?>)
      System.out.println("iOb2 es instancia de Gen");

    System.out.println( );

    // Ve si strOb2 es un Gen2.
    if (strOb2 instanceof Gen2<?>)
      System.out.println("strOb2 es instancia de Gen2");

    // Ve si strOb2 es un Gen.
    if(strOb2 instanceof Gen<?>)
      System.out.println("strOb2 es instancia de Gen");

    System.out.println( );

    // Ve si iOb es una instancia de Gen2. No lo es.
    if(iOb instanceof Gen2<?>)
      System.out.println("iOb es instancia de Gen2");

    // Ve si iOb es una instancia de Gen. Si lo es.
    if (iOb instanceof Gen<?>)
      System.out.println("iOb es instancia de Gen");

    // Lo siguiente no puede ser compilado porque
    // la información de tipos parametrizados no existe en tiempo de ejecución
//    if(iOb2 instanceof Gen2<Integer>)
//    System.out.println("iOb2 es instancia de Gen2<Integer>");
  }
}
```

La salida del programa se muestra a continuación:

```
iOb2 es una instancia de Gen2
iOb2 es una instancia de Gen
```

```
strOb2 es una instancia de Gen2
strOb2 es una instancia de Gen

iOb es una instancia de Gen
```

En este programa, **Gen2** es una subclase de **Gen**, la cual tiene un tipo parametrizado llamado **T**. En el método **main( )**, se crean tres objetos. El primero es **iOb**, el cual es un objeto de tipo **Gen<Integer>**. El segundo es **iOb2**, el cual es una instancia de **Gen2<Integer>**. Finalmente, **strOb2** que es un objeto de tipo **Gen2<String>**.

Entonces, el programa ejecuta las siguientes pruebas con **instanceof** sobre el tipo de **iOb2**:

```
// Ve si iOb2 tiene alguna forma de Gen2.
if (iOb2 instanceof Gen2<?>)
  System.out.println("iOb2 es instancia de Gen2");

// Ve si iOb2 tiene alguna forma de Gen
if (iOb2 instanceof Gen<?>)
  System.out.println("iOb2 es instancia de Gen");
```

Como lo muestra la salida, ambos casos son exitosos. En la primer prueba, **iOb2** se revisa contra **Gen2<?>**. Esta prueba es exitosa simplemente porque confirma que **iOb2** es un objeto de algún tipo de **Gen2**. El uso del comodín permite al operador **instanceof** determinar si **iOb2** es un objeto de cualquier tipo de **Gen2**. El siguiente, **iOb2** se prueba contra **Gen<?>**, el tipo superclase. Esto también es exitoso porque **iOb2** es alguna forma de Gen, la superclase. Las siguientes líneas, en el método **main( )** muestran la misma secuencia (y mismos resultados) para **strOb2**.

A continuación, **iOb**, la cual es una instancia de **Gen<Integer>** (la superclase), se prueba con estas líneas:

```
// Ve si iOb es una instancia de Gen2. No lo es.
if(iOb instanceof Gen2<?>)
  System.out.println("iOb es instancia de Gen2");

// Ve si iOb es una instancia de Gen. Si lo es.
if (iOb instanceof Gen<?>)
  System.out.println("iOb es instancia de Gen");
```

La primera **condición** falla porque **iOb** no es de ningún tipo de **Gen2**. La segunda condición es exitosa porque **iOb** es de algún tipo de **Gen**.

Ahora, observemos más de cerca de las líneas comentadas:

```
   //  Lo siguiente no puede ser compilado porque
   //  la información de tipos parametrizados no existe en tiempo de ejecución
//     if(iOb2 instanceof Gen2<Integer>)
//     System.out.println("iOb2 es instancia de Gen2<Integer>");
```

Como los comentarios lo indican, estas líneas no pueden ser compiladas porque intentan comparar **iOb2** con un tipo específico de **Gen2**, en este caso, **Gen2<Integer>**. Recuerde, que no hay información de tipos parametrizados disponible en tiempo de ejecución. Además, no hay forma de que el operador **instanceof** sepa si **iOb2** es una instancia de **Gen2<Integer>** o no.

Conversión de tipos

Se puede convertir una instancia de una clase genérica en otra sólo si las dos son compatibles de alguna forma y sus argumentos de tipos son los mismos. Por ejemplo, en el programa anterior, esta conversión es correcta:

```
(Gen<Integer>) iOb2  // es correcto
```

debido a que **iOb2** es una instancia de **Gen<Integer>**. Pero, la conversión:

```
(Gen<Long>) iOb2 // es incorrecta
```

no es correcta porque **iOb2** no es una instancia de **Gen<Long>**

Sobreescritura de métodos en clases con tipos parametrizados

Un método en una clase con tipos parametrizados puede ser sobrescrito como cualquier otro método. Por ejemplo, en el siguiente programa el método **getob( )** es sobrescrito:

```
// Sobrescribe un método con tipos parametrizados en una clase
con tipos parametrizados
class Gen<T> {
  T ob;  // declara un objeto de tipo T

  // Pasa al constructor una referencia a
  // un objeto de tipo T.
  Gen(T o) {
    ob = o;
  }

  // Devuelve ob.
  T getob ( ) {
    System.out.print("Llamada al método getob( ) de Gen: ");
    return ob;
  }
}

// Una subclase de Gen que sobrescribe getob( ).
class Gen2<T> extends Gen<T> {

  Gen2(T o) {
    super(o);
  }

  // Sobrescribe getob( ).
  T getob ( ) {
    System.out.print("El método getob( ) de Gen2: ");
    return ob;
  }
}

  // Sobreescritura de un método con tipos parametrizados
  class SobrescrituraDemo {
    public static void main(String args[]) {
```

```
    // Crea un objeto Gen para Integer.
    Gen<Integer> iOb = new Gen<Integer> (88);

    // Crea un objeto Gen2 para Integer.
    Gen2<Integer> iOb2 = new Gen2<Integer>(99);

    // Crea un objeto Gen2 para String.
    Gen2<String> strOb2 = new Gen2<String> ("Prueba de tipos parametrizados");

    System.out.println(iOb.getob( ));
    System.out.println(iOb2.getob( ));
    System.out.println(strOb2.getob( ));
  }
}
```

La salida se muestra a continuación:

```
El método getob( ) de Gen: 88
El método getob( ) de Gen2: 99
El método getob( ) de Gen2: Prueba de tipos parametrizados
```

Cómo están implementados los tipos parametrizados

Usualmente, no es necesario saber los detalles acerca de cómo el compilador de Java transforma el código fuente en código objeto. Sin embargo, en el caso de los tipos parametrizados, es importante entender de manera general el proceso debido a que explica por qué las características de tipos parametrizados funcionan de la manera en que lo hacen – y por qué su comportamiento es, en algunas ocasiones un tanto sorprendente. Por esta razón, es necesario comentar brevemente cómo los tipos parametrizados están implementados en Java.

Una restricción importante que controla la forma en que los tipos parametrizados fueron agregados a Java fue la necesidad de mantener la compatibilidad con las versiones previas de Java. Dicho de forma simple, el código con tipos parametrizados tiene que ser compatible con el código pre-existente de tipos no parametrizados. Cualquier cambio en la sintaxis del lenguaje de Java o de la JVM, tuvo que evitar la ruptura del código anterior. La forma en que Java implementa los tipos parametrizados satisfaciendo esta restricción es a través del uso de la técnica de la *cancelación*.

En general, así es como la cancelación funciona. Cuando el código de Java se compila, toda la información de tipos parametrizados se elimina (cancela). Esto significa que se reemplaza el parámetro de tipo con el tipo correspondiente, el cual es **Object** si no hay tipos especificados explícitamente, y luego se aplican cambios de tipos (como se determinó en los argumentos de tipo) para mantener la compatibilidad de tipos con los tipos especificados por los argumentos. El compilador también implementa este tipo de compatibilidad. Esta estrategia de tipos parametrizados significa que no existen parámetros de tipo en tiempo de ejecución. Son simplemente un mecanismo de codificación.

La mejor forma de entender cómo trabaja la técnica de la cancelación es revisar las siguientes dos clases:

```
// Aquí, por omisión T es reemplazada por Object
class Gen<T> {
  T ob;  // aquí, T será reemplazada por Object

  Gen(T o) {
  ob = o;
  }

  // Devuelve ob.
  T getob ( ) {
    return ob;
  }
}

// Aquí, T es delimitado por String.
class GenStr<T extends String> {
  T str; // aquí, T será reemplazada por String

    GenStr(T o) {
  str = o;
}

  T getstr( ) { return str; }
}
```

Después de que estas dos clases son compiladas, la **T** en **Gen** será reemplazado por **Object**. La **T** en **GenStr** será reemplazada por **String**. Se puede confirmar esto ejecutando **javap** sobre las clases compiladas. El resultado se muestra a continuación:

```
class Gen extends java.lang.Object{
    java.lang.Object ob;
    Gen(java.lang.Object);
    java.lang.Object getob();
}

class GenStr extends java.lang.Object{
    java.lang.String str;
    GenStr (java.lang.String);
    java.lang.String getstr();
}
```

Dentro del código de **Gen** y **GenStr**, la conversión de tipos se utiliza para asegurar la tipificación correcta. Por ejemplo, esta secuencia:

```
Gen<Integer> iOb = new Gen<Integer>(99);

int x = iOb.getob();
```

será compilada como si hubiera sido escrita así:

```
Gen iOb = new Gen(99) ;

int x = (Integer) iOb.getob();
```

Debido a la técnica de la cancelación, algunas cosas funcionan un poco diferente de lo que se podría pensar. Por ejemplo, considere este pequeño programa que crea dos objetos de tipos parametrizados de la clase **Gen**:

```
class GenTypeDemo {
  public static void main(String args[]) {
    Gen<Integer> iOb = new Gen<Integer> (99);
    Gen<Float> fOb = new Gen<Float>(102.2F);

    System.out.println(iOb.getClass().getName());
    System.out.println(fOb.getClass().getName());
  }
}
```

La salida de este programa se muestra a continuación:

```
Gen
Gen
```

Como se puede ver, el tipo tanto de **iOb** como de **fOb** es **Gen**, no **Gen<Integer>** y **Gen<Float>** como se podría esperar. Recuerde, todos los tipos parametrizados son eliminados durante la compilación. En tiempo de ejecución, sólo existen tipos en bruto.

Métodos puente

Ocasionalmente, el compilador necesitará agregar un *método puente* a una clase para gestionar situaciones en las cuales la técnica de cancelación aplicada a un método sobrescrito en una subclase no produce la misma cancelación que el método en la superclase. En este caso, se genera un método que utiliza al tipo cancelado de la superclase, y este método llama al método que tiene el tipo cancelado especificado por la subclase. Por supuesto, los métodos puente sólo ocurren a nivel de bytecode, no son vistos por el programador y no están disponibles para su uso.

Aunque los métodos puente no son algo que normalmente deba preocuparnos, es educativo ver la situación en la cual se generan. Considere el siguiente programa:

```
// Una situación en que se crea un método puente
class Gen<T> {
  T ob;  // declara un objeto de tipo T

  // Pasa al constructor una referencia a
  // un objeto de tipo T.
  Gen(T o) {
    ob = o;
  }

  // Devuelve ob.
  T getob () {
    return ob;
  }
}

// Una subclase de Gen.
class Gen2 extends Gen<String> {
```

```
  Gen2(String o) {
    super (o);
  }

  // Sobreescritura de getob().
  String getob() {
    System.out.print("Se llama al método String getob(): ");
    return ob;
  }
}

// Demuestra la situación que requiere un método puente.
class BridgeDemo {
  public static void main(String args[]) {

    // Crea un objeto Gen2 para String
    Gen2 strOb2 = new Gen2("Prueba de tipos parametrizados");

    System.out.println(strOb2ogetob()) ;
  }
}
```

En el programa, la subclase **Gen2** extiende de **Gen**, pero utiliza **String** como parámetro de tipo para **Gen**:

```
class Gen2 extends Gen<String> {
```

Además, dentro de **Gen2**, el método **getob( )** se sobrescribe definiendo **String** como su tipo de retorno:

```
// Sobreescritura de getob().
String getob() {
  System.out.print("Se llama al método String getob(): ");
  return ob;
}
```

Todo esto es perfectamente aceptable. El único problema es que a causa del tipo cancelado, la forma esperada de **getob( )** será:

```
Object getob(){//...
```

Para gestionar este problema, el compilador genera un método puente con la firma anterior que llama a la versión del método con **String**. De esta forma, si se examina el archivo de la clase **Gen2** utilizando **javap**, se verán los siguientes métodos:

```
class Gen2 extends Gen{
    Gen2(java.lang.String);
    java.lang.String getob();
    java.lang.Object getob(); // método puente
}
```

Como se puede ver, el método puente ha sido incluido (el comentario fue agregado por el autor, y no por **javap**).

Existe un último punto a resaltar sobre los métodos puente. Note que la única diferencia entre los dos métodos **getob( )** está en el tipo de retorno. Normalmente, esto causaría un error,

pero debido a que esto no está presente en el código fuente, no se genera ningún problema y la ejecución se realiza de forma correcta por la JVM.

Errores de ambigüedad

La inclusión de tipos parametrizados permite el surgimiento de un nuevo tipo de error del cual debemos tener cuidado: *ambigüedad*. Los errores de ambigüedad ocurren cuando la técnica de cancelación causa dos declaraciones aparentemente distintas de tipos parametrizados para resolver el mismo tipo cancelado, ocasionando un conflicto. A continuación hay un ejemplo que involucra sobrecarga de métodos:

```
// Ambigüedad causada por la técnica de la cancelación aplicada a
// métodos sobrecargados
class MyGenClass<T, V> {
  T ob1;
  V ob2;

  // . . .

  // Estos dos métodos sobrecargados son ambiguos
  // y no compilarán.
  void set(T o) {
    ob1 = o;
  }

  void set(V o) {
    ob2 = o;
  }
}
```

Note que **MyGenClass** declara dos tipos parametrizados: **T** y **V**. Dentro de **MyGenClass**, se hace un intento para sobrecargar **set( )** con base a los parámetros de tipo T y V. Esto parece razonable porque **T** y **V** parecen ser tipos diferentes. Sin embargo, existen dos problemas de ambigüedad aquí.

Primero, debido a la forma en que **MyGenClass** está escrita, no hay requerimientos de que **T** y **V** sean diferentes tipos. Por ejemplo, es perfectamente correcto (en principio) construir un objeto **MyGenClass** como se muestra a continuación:

```
MyGenClass<String, String> obj = new MyGenClass<String, String>()
```

En este caso, ambos **T** y **V** serán reemplazados por **String**. Esto hace que ambas versiones de **set( )** sean idénticas, lo cuál, por supuesto, es un error.

El segundo y más importante problema es que el tipo cancelado de **set( )** reduce ambas versiones a lo siguiente:

```
void set(Object o) { // ...
```

De esta forma, la sobrecarga del método **set( )** intentada en **MyGenClass** es intrínsicamente ambigua.

Los errores de ambigüedad pueden ser difíciles de arreglar. Por ejemplo, si se conoce que **V** será siempre de tipo **String**, se puede intentar arreglar **MyGenClass** escribiendo nuevamente su declaración como se muestra a continuación:

```
class MyGenClass<T, V extends String> { // casi correcto
```

Este cambio causa que **MyGenClass** compile, e incluso se pueden instanciar objetos como el que se muestra aquí:

```
MyGenClass<Integer, String> x = new MyGenClass<Integer, String>();
```

Esto funciona debido a que Java puede determinar exactamente a cuál método llamar. Sin embargo, la ambigüedad vuelve cuando se intenta esta línea:

```
MyGenClass <String, String> x = new MyGenClass<String, String>();
```

En este caso, dado que tanto **T** como **V** son **String**, ¿cuál versión de **set( )** será llamada?

Francamente, en el ejemplo anterior, sería mucho mejor utilizar dos métodos con nombres separados, en lugar de intentar sobrecargar **set( )**. Frecuentemente, la solución para la ambigüedad envuelve la reestructuración del código, porque frecuentemente la ambigüedad significa que se tiene un error conceptual en el diseño.

Restricciones de los tipos parametrizados

Hay algunas restricciones que es necesario tener en mente cuando se utilizan tipos parametrizados. Éstas involucran la creación de objetos de un parámetro de tipo, miembros estáticos, excepciones, y arreglos. Cada una se examina a continuación.

Los tipos parametrizados no pueden ser instanciados

No es posible crear una instancia de un parámetro de tipo. Por ejemplo, considere el siguiente caso:

```
//No se puede crear una instancia de T
class Gen<T> {
  T ob;
  Gen( ) {
    ob = new T( ); // error
  }
}
```

Es incorrecto intentar crear una instancia de **T**. La razón debería ser fácil de entender: porque **T** no existe en tiempo de ejecución, ¿cómo sabría el compilador qué tipo de objeto crear? Recuerde que la cancelación elimina todos parámetros de tipo durante el proceso de compilación.

Restricciones en miembros estáticos

Los miembros **static** de una clase no pueden utilizar a los parámetros de tipo de la clase. Por ejemplo, todos los miembros **estáticos** de esta clase son incorrectos:

```
class Wrong<T>{
  //Error, no puede haber variables estáticas de tipo T.
  static T ob;

  //Error, no puede haber métodos estáticos de tipo T.
  static T getob( ) {
```

```
    return ob;
  }

  // Error, no puede haber métodos estáticos que accedan a objetos
  // de tipo T.
  static void showob( ) {
    System.out.println(ob) ;
  }
}
```

Aunque no se pueden declarar miembros **estáticos** que utilicen parámetros de tipo declarados por la clase, se *pueden* declarar métodos **estáticos** de tipos parametrizados, que definan sus propios tipos de parámetros, tal como se hizo anteriormente en este capítulo.

Restricciones en arreglos con tipos parametrizados

Existen dos restricciones importantes de los tipos parametrizados que aplican a los arreglos. En primer lugar, no se puede instanciar un arreglo cuyo tipo base es un parámetro de tipo. En segundo lugar, no se puede crear un arreglo como una referencia a un tipo parametrizado específico. El siguiente programa muestra ambas situaciones:

```
// Tipos parametrizados y arreglos
class Gen<T extends Number> {
  T ob;

  T vals[]; // correcto

  Gen(T o, T[] nums) {
    ob = o;

    // Esta sentencia es incorrecta
    // vals = new T[10]; //no se puede crear un arreglo de T

    // Pero, esta sentencia es correcta.
    vals = nums; // es correcto asignar una referencia a un arreglo existente
  }
}

class GenArrays {
  public static void main(String args[]) {
    Integer n[] = { 1, 2, 3, 4, 5 };

    Gen<Integer> iOb = new Gen<Integer> (50, n);

    // No se puede crear un arreglo con una referencia a un tipo
    parametrizado específico
    // Gen<Integer> gens[] = new Gen<Integer> [10]; // error

    // Esto es correcto
    Gen<?> gens[] = new Gen<?> [10]; // correcto
  }
}
```

Como lo muestra el programa, es válido declarar una referencia a un arreglo de tipo **T**, como lo hace la siguiente línea:

```
T vals[]; // correcto
```

Pero no se puede hacer instancia un arreglo de tipo **T**, como lo muestra la siguiente línea comentada.

```
// vals = new T[10]; //no se puede crear un arreglo de tipo T
```

La razón por la cual no se puede crear un arreglo de tipo **T** es porque **T** no existe en tiempo de ejecución, entonces, no existe una forma para el compilador de saber qué tipo de arreglo tiene que crear.

Sin embargo, se puede pasar una referencia a un arreglo de tipo compatible a **Gen( )** cuando un objeto es creado y asigna esa referencia a **vals**, como el programa lo hace en esta línea:

```
vals = nums; // es correcto asignar una referencia a un arreglo existente.
```

Esto funciona porque el arreglo que se pasa a **Gen** tiene un tipo conocido, el cual será el mismo tipo que **T** en el momento en que el objeto sea creado.

Dentro del método **main( )**, note que no se puede declarar un arreglo de referencias a un tipo parametrizado específico. Esto se muestra a continuación:

```
// Gen<Integer> gens[] = new Gen<Integer>[10];  // error
```

No compilará. Los arreglos de tipos parametrizados simplemente no están permitidos, debido a que podrían causar la pérdida de seguridad en el manejo de tipos.

Se *puede* crear un arreglo como referencia a un tipo parametrizados si se utilizan comodines, como se muestra a continuación:

```
Gen<?> gens[] = new Gen<?>[10];   // correcto
```

Esta estrategia es mejor que utilizar arreglos de tipos en bruto, porque al menos algunas validaciones de tipos serán realizadas.

Restricciones en excepciones con tipos parametrizados

Una clase con tipos parametrizados no puede extender de **Throwable**. Esto significa que no se pueden crear clases para excepciones genéricas.

Comentarios adicionales sobre tipos parametrizados

Los tipos parametrizados son una poderosa extensión de Java debido a que modernizan la creación de tipos seguros y código reutilizable. Aunque la sintaxis de los tipos parametrizados parece ser abrumadora al inicio, se vuelve natural después de utilizarla por un tiempo. El código con tipos parametrizados será parte del futuro para todos los programadores en Java.

PARTE II

La biblioteca de Java

15 CAPÍTULO

Gestión de cadenas

En el Capítulo 7 se realizó una breve introducción a la gestión de cadenas en Java. En este capítulo trataremos este tema con mayor detalle. Como ocurre en la mayoría de los lenguajes de programación, en Java una *cadena* es una secuencia de caracteres. Pero, al contrario que muchos otros lenguajes que implementan las cadenas como arreglos de caracteres, Java las implementa como objetos del tipo **String**.

La incorporación de las cadenas como objetos en Java permite proporcionar un conjunto completo de características que facilitan su manipulación. Por ejemplo, Java tiene métodos para comparar dos cadenas, buscar subcadenas, concatenar cadenas o intercambiar mayúsculas y minúsculas dentro de una cadena. Además, los objetos de tipo **String** se pueden construir de diferentes maneras, facilitando la obtención de una cadena cuando se necesita.

Sin embargo, ocurre algo hasta cierto punto inesperado: cuando se crea un objeto de tipo **String**, se está creando una cadena que no se puede modificar; es decir, una vez creado un objeto **String**, no se pueden cambiar los caracteres que lo conforman. A primera vista, esta puede parecer una restricción muy seria. Sin embargo, no es este el caso. Aún se pueden llevar a cabo todo tipo de operaciones con cadenas. La diferencia es que cada vez que se necesite una versión alterada de una cadena existente, se debe crear un nuevo objeto **String** que contenga las modificaciones. La cadena original se queda como estaba. Esto se hace así porque las cadenas fijas e inmutables se pueden implementar mucho más eficientemente que las que cambian. Para los casos en que se desee una cadena modificable, Java proporciona dos opciones: **StringBuffer** y **StringBuilder**. Los objetos de estas dos clases contienen cadenas que se pueden modificar aún después de ser creadas.

Las clases **String**, **StringBuffer** y **StringBuilder** están definidas en el paquete **java.lang**, por lo que se encuentran disponibles para todos los programas automáticamente. Todas están declaradas como **final**, lo que significa que no se pueden crear subclases a partir de ellas. Esto permite ciertas optimizaciones que mejoran el rendimiento en las operaciones con cadenas más comunes. Las tres clases implementan la interfaz **CharSequence**.

Una cosa más: las cadenas que son objetos de tipo **String** son inmodificables, lo que significa que el contenido de la instancia **String** no se puede cambiar después de crearse. Sin embargo, una variable declarada como referencia **String** se puede cambiar en cualquier momento para que apunte a otro objeto **String**.

Los constructores String

La clase **String** soporta varios constructores. Para crear una **cadena** vacía se puede utilizar el constructor por omisión. Por ejemplo,

```
String s = new String();
```

creará una instancia de **String** sin ningún carácter en ella.

A menudo se desea crear cadenas con valores iniciales. La clase **String** proporciona diferentes constructores para ello. Para crear un objeto **String** inicializado con un arreglo de caracteres, se puede usar el siguiente constructor:

String(char *chars*[ ])

Por ejemplo:

```
char chars[] = {'a', 'b', 'c'};
String s = new String(chars);
```

Este constructor inicializa **s** con la cadena"abc".

Se puede especificar un subrango de un arreglo de caracteres como inicializador utilizando el siguiente constructor:

```
String (char chars[ ], int indiceInicio, int numeroCaracteres)
```

Aquí, *indiceInicio* especifica el índice en que comienza el subrango, y *numeroCaracteres* es el número de caracteres a emplear. Por ejemplo:

```
char chars[] = { 'a', 'b', 'c', 'd', 'e', 'f' };
String s = new String(chars, 2, 3);
```

Esto inicializa **s** con los caracteres **cde**.

Se puede construir un objeto **String** que contenga la misma secuencia de caracteres que otro utilizando este constructor:

String (String *objetoString*)

Aquí, *objetoString* es un objeto de tipo **String**. Por ejemplo:

```
// Construir un String a partir de otro.
class MakeString {
  public static void main(String args[] ) {
    char c[] = {'J', 'a', 'v', 'a'};
    String s1 = new String(c);
    String s2 = new String(s1);

    System.out.println(s1);
    System.out.println(s2);
  }
}
```

La salida de este programa es como sigue:

```
Java
Java
```

Como se puede observar, **s1** y **s2** contienen la misma cadena.

Aunque el tipo **char** de Java usa 16 bits para representar el conjunto de caracteres Unicode, el formato típico para las cadenas en Internet usa arreglos de bytes (8 bits) con el conjunto de caracteres ASCII.

Dado que las cadenas ASCII de 8 bits son comunes, la clase **String** proporciona constructores que inicializan una cadena a partir de un arreglo de **bytes**. Sus formas se muestran a continuación:

String(byte *caracteresAscii*[ ])
String(byte *caracteresAscii*[ ], int *indiceInicial*, int *numeroCaracteres*)

Aquí, *caracteresAscii* especifica el arreglo de bytes. La segunda forma permite especificar un subrango. En cada uno de estos constructores, la conversión de byte a carácter se realiza usando la codificación de caracteres por omisión de la plataforma. El siguiente programa ilustra estos constructores:

```
// Construcción de una cadena a partir de un arreglo de caracteres.
class SubStringCons {
  public static void main(String args[]) {
    byte ascii[] = {65, 66, 67, 68, 69, 70 };

    String s1 = new String(ascii);
    System.out.println(s1);

    String s2 = new String(ascii, 2, 3);
    System.out.println(s2);
  }
}
```

La salida de este programa es:

```
ABCDEF
CDE
```

También se definen versiones extendidas de los constructores byte-a-cadena, en los que se puede especificar la codificación de caracteres que determina cómo se convierten los bytes en caracteres. Sin embargo, la mayoría de las veces se utiliza la codificación proporcionada por la plataforma.

NOTA *Los contenidos del arreglo se copian cada vez que se crea un objeto String a partir de un arreglo. Si se modifican los contenidos del arreglo después de creada la cadena, el objeto String no se modifica.*

Es posible construir un objeto de tipo **String** a partir de un objeto **StringBuffer** utilizando el constructor:

String (StringBuffer *strBufObj*)

También es posible construir un objeto **String** a partir de un objeto **StringBuilder** con el constructor:

String (StringBuilder *strBuildObj*)

El siguiente constructor permite utilizar el conjunto de caracteres extendido Unicode:

String (int *codigos*[ ], int *indiceInicial*, int *numeroCaracteres*)

Aquí *codigos* es un arreglo que contiene los códigos Unicode. La cadena resultante es construida dentro del rango que comienza en *indiceInicial* y hasta *numeroCaracteres*.

Java SE 6 añade además la posibilidad de construir una **cadena** a partir de un objeto del tipo **Charset**.

NOTA *En el Capítulo 16 se presenta con mayor detalle Unicode y cómo es gestionado por Java.*

Longitud de una cadena

La longitud de una cadena es el número de caracteres que contiene. Para obtener este valor se usa el método **length( )**, mostrado a continuación:

int length( )

El siguiente fragmento de código imprime"3", pues la cadena **s** tiene tres caracteres:

```
char chars[] = {'a', 'b', 'c'};
String s = new String(chars);
System.out.println(s.length());
```

Operaciones especiales con cadenas

Dado que las cadenas son una parte común e importante de la programación, Java proporciona dentro de sus sintaxis un soporte especial para diversas operaciones con cadenas. Estas operaciones incluyen la creación automática de nuevas instancias **String** a partir de literales de cadena, la concatenación de múltiples objetos **String** mediante el uso del operador **+**, así como la conversión de otros tipos de datos en una representación de tipo cadena. Hay métodos explícitos disponibles para realizar todas estas funciones, pero Java lo hace automáticamente para facilitar el trabajo del programador, y también para añadir claridad.

Literales de cadena

Los ejemplos anteriores muestran cómo crear explícitamente una instancia **String** a partir de un arreglo de caracteres mediante el operador **new**. Sin embargo, hay un modo más fácil de hacer esto usando un literal de cadena. Por cada literal de cadena que haya en un programa, Java automáticamente construye un objeto **String**. Por ello, se puede usar un literal de cadena para inicializar un objeto **String**. Por ejemplo, el siguiente fragmento de código crea dos cadenas equivalentes:

```
char chars[] = {'a', 'b','c'};
String s1 = new String(chars);

String s2 = "abc"; // utiliza un literal de cadena
```

Puesto que se crea un objeto **String** para cada literal de cadena, se puede utilizar una literal de cadena en cualquier sitio en el que se pueda usar un objeto **String**. Por ejemplo, se puede llamar directamente a los métodos con una cadena entre comillas como si fuera una referencia

a un objeto, tal como muestra la siguiente sentencia, que llama al método **length( )** sobre la cadena "abc". Como es de esperar, imprime "3":

```
System.out.println("abc".length());
```

Concatenación de cadenas

En general, Java no permite aplicar operadores a los objetos **String**. La única excepción a esta regla es el operador **+**, que concatena dos cadenas produciendo un objeto **String** como resultado. Esto permite yuxtaponer una serie de operaciones **+**. Por ejemplo, el siguiente fragmento concatena tres cadenas:

```
String edad = "9";
String s = "Ella tiene" + edad + " años.";
System.out.println(s);
```

Esto muestra la cadena "Ella tiene 9 años."

Un uso práctico de la concatenación de cadenas se da cuando se crean cadenas muy largas. En lugar de permitir que cadenas muy largas embarullen el código fuente, se pueden romper en trozos más pequeños y usar el operador **+** para concatenarlas. Por ejemplo:

```
// Uso de la concatenación para evitar líneas largas.
class ConCat {
  public static void main(String args[]) {
    String longStr = "Ésta podría haber sido " +
      "una línea muy larga que habría saltado " +
      "a las siguientes líneas. Pero la " +
      "concatenación de cadenas lo evita.";

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

Concatenación de cadenas con otros tipos de datos

Se puede concatenar cadenas con otros tipos de datos. Por ejemplo, considere esta versión ligeramente distinta del ejemplo anterior:

```
int edad = 9;
String s = "Ella tiene" + edad + " años.";
System.out.println(s);
```

En este caso, **edad** es de tipo **int** en lugar de otro objeto **String**, pero la salida del código es la misma que antes. Esto debido a que el valor **int** de **edad** se convierte automáticamente a su representación de cadena dentro de un objeto **String**; entonces esta cadena se concatena como antes. El compilador convertirá un operando en su cadena equivalente siempre que el otro operando del operador **+** sea una instancia de **String**.

Sin embargo, hay que tener cuidado al mezclar otros tipos de operaciones con expresiones de concatenación de cadenas, ya que se puede obtener resultados sorprendentes. Consideremos el siguiente fragmento de código:

```
String s = "cuatro: " + 2 + 2;
System.out.println(s);
```

La salida es:

```
cuatro: 22
```

en vez de:

```
cuatro: 4
```

que probablemente era el resultado esperado. La precedencia de operadores hace que en primer lugar se concatene la cadena "cuatro:" con la cadena equivalente del primer número 2. Después, se concatena este resultado con la cadena equivalente del segundo número 2. Para realizar primero la suma de enteros hay que utilizar paréntesis:

```
String s = "cuatro: " + (2 + 2);
```

Ahora **s** contiene la cadena `"cuatro: 4"`.

Conversión de cadenas y toString( )

Cuando Java convierte datos en su representación de cadena durante la concatenación, lo hace llamando a una versión sobrecargada del método de conversión de cadenas **valueOf( )** definido por **String**. El método **valueOf( )** está sobrecargado para todos los tipos simples y para el tipo **Object**. Para los tipos primitivos, **valueOf( )** devuelve una cadena que contiene el texto legible equivalente del valor con que se le llama. Para objetos, **valueOf( )** llama al método **toString( )** sobre ese objeto. Analizaremos **valueOf( )** con más detalle más adelante en este capítulo. Aquí vamos a examinar el método **toString( )**, porque es la manera de obtener la representación en cadena de objetos de clases creadas por el programador.

Todas las clases implementan el método **toString( )** porque este método está definido en la clase **Object**. Sin embargo, la implementación por omisión de **toString( )** raramente es suficiente. Para la mayoría de las clases importantes creadas por el programador, será deseable sobrescribir el método **toString( )** y proporcionar nuestras propias representaciones en forma de cadena. Afortunadamente, esto es fácil de hacer. El método **toString( )** tiene esta forma general:

String toString( )

Para implementar **toString( )**, basta simplemente con devolver un objeto **String** que contenga la cadena legible que describa apropiadamente al objeto de la clase.

Al sobrescribir **toString( )** en las clases creadas por el programador, se permite a las cadenas resultantes integrarse totalmente en el entorno de programación de Java. Por ejemplo, se pueden usar en las sentencias **print( )** y **println( )**, así como en expresiones de concatenación. El siguiente programa muestra esto sobrescribiendo **toString( )** para la clase Box:

```
// Sobrescribir toString() para la claseBox.
class Box {
  double anchura;
  double altura;
  double profundidad;

  Box(double w, double h, double d) {
    anchura = w;
    altura = h;
    profundidad =d;
  }
```

```
  public String toString() {
    return "Las dimensiones son " + anchura + " por " +
      profundidad + " por " + altura + ".";
  }
}

class toStringDemo {
  public static void main(String args[]) {
    Box b = new Box(10, 12, 14);
    String s = "Box b: " + b; // concatena al objeto Box
    System.out.println(b); // convierte Box a cadena
    System.out.println(s);
  }
}
```

La salida de este programa es:

```
Las dimensiones son 10.0 por 14.0 por 12.0
Box b: Las dimensiones son 10.0 por 14.0 por 12.0
```

Como se ve, el método **toString( )** de la clase **Box** es llamado automáticamente cuando se usa un objeto **Box** en una expresión de concatenación o en una llamada a **println( )**.

Extracción de caracteres

La clase **String** proporciona diferentes modos de extraer caracteres de un objeto **String**. A continuación examinaremos cada uno de ellos. Aunque los caracteres que componen una cadena dentro de un objeto **String** no se pueden indexar como si fueran un arreglo de caracteres, muchos de los métodos de **String** emplean un índice (o desplazamiento) dentro de la cadena para su funcionamiento. Al igual que los arreglos, los índices de cadenas comienzan en cero.

charAt( )

Para extraer un único carácter de un objeto **String**, se puede hacer referencia directamente a un carácter individual mediante el método **charAt( )**. Tiene la siguiente forma general:

char charAt(int *donde*)

Aquí, *donde* es el índice del carácter que se quiere obtener. El valor de *donde* debe ser no negativo y especificar una posición dentro de la cadena. **charAt( )** devuelve el carácter en la posición especificada. Por ejemplo,

```
char ch;
ch = "abc".charAt(1);
```

asigna el valor "**b**"a **ch**.

getChars( )

Si se necesita extraer más de un carácter a la vez, se puede usar el método **getChars( )**, el cual tiene la forma general:

void getChars(int *posInicial*, int *posFinal*, char *destino*[ ], int *posDestino*)

Donde *posInicial* especifica el índice donde comienza la subcadena y *posFinal* la posición siguiente a aquella en que se desea termine la subcadena. Así, la subcadena contiene los caracteres desde *posInicial* hasta *posFinal*-1. El arreglo que reciben los caracteres es el especificado por *destino*, y el índice dentro de *destino* a partir del cual se copia la subcadena es indicado con *posDestino*. Hay que tener cuidado de que el arreglo de *destino* sea lo suficientemente grande como para contener todos los caracteres de la subcadena especificada.

El siguiente programa muestra el uso de **getChars( ):**

```
class getCharsDemo {
  public static void main(String args[]) {
    String s = "Esta es una demo del método getChars.";
    int start = 12;
    int end = 16;
    char buf[] = new char[end - start];

    s.getChars(start, end, buf, 0);
    System.out.println(buf);
  }
}
```

He aquí la salida de este programa:

```
demo
```

getBytes( )

Existe una alternativa a **getChars( )** que almacena los caracteres en un arreglo de bytes. Este método se llama **getBytes( )**, y utiliza las conversiones carácter a byte proporcionadas por omisión por la plataforma. Su forma más simple es:

byte[ ] getBytes( )

También están disponibles otras formas de **getBytes( )**. La mayor utilidad de **getBytes( )** se da cuando al exportar un valor **String** a un entorno que no soporta los caracteres Unicode de 16 bits. Por ejemplo, la mayoría de los protocolos de Internet y formatos de archivos de texto utilizan el código ASCII de 8 bits.

toCharArray( )

Si se desea convertir todos los caracteres de un objeto **String** a un arreglo de caracteres, el modo más fácil de hacerlo es llamando al método **toCharArray( )**. Este método devuelve un arreglo de caracteres con la cadena completa. Su forma general es:

char[ ] toCharArray( )

Esta función se proporciona para facilitar la tarea del programador, pues siempre es posible conseguir el mismo resultado utilizando **getChars( )**.

Comparación de cadenas

La clase **String** incluye diferentes métodos para comparar cadenas o subcadenas dentro de cadenas. A continuación examinaremos cada una de ellas.

equals( ) y equalsIgnoreCase( )

Para comparar la igualdad de dos cadenas se utiliza el método **equals( )**, el cual tiene la siguiente forma general:

boolean equals(Object *str*)

Aquí, *str* es el objeto **String** que se compara con el objeto **String** que llama al método. Devuelve **true** si las cadenas contienen los mismos caracteres en el mismo orden, y **false** en caso contrario. La comparación distingue mayúsculas de minúsculas.

Para hacer una comparación que ignore las diferencias entre mayúsculas y minúsculas, podemos utilizar **equalsIgnoreCase( )**, el cual, al comparar dos cadenas, considera a los caracteres de **A-Z** iguales a los caracteres **a-z**. El método tiene la forma general:

boolean equalsIgnoreCase(String *str*)

Aquí, *str* es el objeto **String** que se compara con el objeto **String** que llama al método. Devuelve **true** si las cadenas contienen los mismos caracteres en el mismo orden, y **false** si no.

He aquí un ejemplo que muestra el uso de **equals( )** y **equalsIgnoreCase( ):**

```
// Ejemplo con equals() y equalsIgnoreCase().
class equalsDemo {
  public static void main(String args[]) {
    String s1 = "Hola";
    String s2 = "Hola";
    String s3 = "Adiós";
    String s4 = "HOLA";
    System.out.println(s1 + " equals " + s2 + " -> " +
                       s1.equals(s2));
    System.out.println(s1 + " equals " + s3 + " -> " +
                       s1.equals(s3));
    System.out.println(s1 + " equals " + s4 + " -> " +
                       s1.equals(s4));
    System.out.println(s1 + " equalsIgnoreCase " + s4 + " -> " +
                       s1.equalsIgnoreCase(s4));
  }
}
```

La salida del programa se muestra a continuación:

```
Hola equals Hola -> true
Hola equals Adiós -> false
Hola equals HOLA -> false
Hola equalsIgnoreCase HOLA -> true
```

regionMatches( )

El método **regionMatches( )** compara una región específica dentro de una cadena con otra región específica dentro de otra cadena. Hay una forma sobrecargada del método que permite ignorar la diferencia entre mayúsculas y minúsculas en tales comparaciones. Las formas generales de estos dos métodos son:

boolean regionMatches(int *posInicial*, String *str2*,
int *posInicialStr2*, int *numCaracts*)

boolean regionMatches(boolean *ignorarCaso*,
int *posInicial*, String *str2*,
int *posInicialStr2*, int *numCaracts*)

En ambas versiones, *posInicial* especifica el índice en que comienza la región dentro del objeto **String** que llama al método. El objeto **String** comparado se especifica en *str2*. El índice en que comienza la comparación dentro de *str2* se especifica en *posInicialStr2*. La longitud de la subcadena comparada se pasa en *numCaracts*. En la segunda versión, si *ignorarCaso* es **true**, se ignora la diferencia entre mayúsculas y minúsculas en los caracteres.

startsWith( ) y endsWith( )

La clase **String** define dos rutinas que son formas más o menos especializadas de **regionMatches( )**. El método **startsWith( )** determina si un objeto **String** dado comienza con una cadena especificada. Análogamente, **endsWith( )** determina si el **String** en cuestión termina con una cadena especificada. Esos métodos tienen las siguientes formas generales:

boolean startsWith(String *str)*
boolean endsWith(String *str*)

Aquí, *str* es la **cadena** que se busca. Si la cadena coincide, se devuelve **true**; de lo contrario, se devuelve **false**. Por ejemplo,

```
"Klostix".endsWith("tix")
```

y

```
"Oscludo".startswith ("Os")
```

devuelven en ambos casos **true**.

Una segunda forma de **startsWith( )**, mostrada a continuación, permite especificar un punto de inicio:

boolean startsWith(String *str*, int *posInicio*)

Aquí, *posInicial* especifica el índice dentro de la cadena que llama al método en el que comenzará la búsqueda. Por ejemplo.

```
"Klostix".startsWith("tix", 4)
```

devuelve **true**.

Comparando equals( ) con el Operador = =

Es importante entender que el método **equals( )** y el operador **==** realizan dos funciones diferentes. Como se acaba de explicar, el método **equals( )** compara los caracteres dentro de un objeto **String**. El operador **==** compara dos referencias de objeto para ver si se refieren a la misma instancia. El siguiente programa muestra cómo dos objetos **String** diferentes pueden contener los mismos caracteres, pero las referencias a estos objetos son distintas.

```
// comparando equals() con el operador ==
class EqualsNotEqualTo {
  public static void main(String args[]) {
    String s1 = "Hola";
    String s2 = new String (s1);
```

```
    System.out.println(s1 + " equals " + s2 + "->" +
                       s1.equals(s2));
    System.out.println(s1 + "==" + s2 + "->" + (s1 == s2));
  }
}
```

La variable **s1** se refiere a la instancia **String** creada por "**Hola**". El objeto al que se refiere **s2** se crea con **s1** como inicializador. Por tanto, los contenidos de ambos objetos **String** son idénticos, pero son objetos distintos. Esto significa que **s1** y **s2** no se refieren a los mismos objetos y por tanto, no son **==**, como se muestra a continuación con la salida del ejemplo anterior:

```
Hola equa1s Hola -> true
Hola == Hola -> false
```


compareTo( )

A menudo no basta simplemente con saber si una cadena es idéntica a otra. En las aplicaciones que requieren ordenar datos se necesita saber si una cadena es *menor*, *igual* o *mayor* que la otra. Una cadena es menor que otra si está delante de ella en orden alfabético. Una cadena es mayor que otra si está después de ella en orden alfabético. El método **compareTo( )** de la clase **String** sirve para esto. Tiene la forma general:

int compareTo(String *str*)

Aquí, *str* es la **cadena** que se compara con el objeto **String** que llama al método. El resultado devuelto por la comparación se interpreta como sigue:

Valor	Significado
Menor que cero	La cadena que llama al método es menor que *str*.
Mayor que cero	La cadena que llama al método es mayor que *str*.
Cero	Ambas cadenas son iguales.

El siguiente programa de ejemplo ordena un arreglo de cadenas, utilizando el método **compareTo( )** para determinar la posición de cada cadena:

```
// Ordenación de cadenas por el método burbuja.
class SortString {
  static String arr[] = {
    "Ahora", "es", "el", "momento", "de", "que", "todos", "los",
    "hombres", "buenos", "vengan", "a", "ayudar", "a", "su", "país"
  };
  public static void main(String args[]) {
    for(int j = 0; j < arr.length; j++) {
      for(int i = j + 1; i < arr.length; i++) {
        if(arr[i].compareTo(arr[j]) < 0) {
          String t = arr[j];
          arr[j] = arr[i] ;
          arr[i] = t;
        }
      }
```

```
      System.out.println(arr[j]);
    }
  }
}
```

La salida de este programa es la siguiente lista de palabras:

```
Ahora
a
a
ayudar
buenos
de
el
es
hombres
los
momento
país
que
su
todos
vengan
```

Como se ve por la salida de este ejemplo, **compareTo( )** toma en cuenta las mayúsculas y las minúsculas. La palabra "Ahora" ha sido listada en primer lugar porque comienza con mayúscula, lo que significa que tiene un valor más bajo en el conjunto de caracteres ASCII.

Para ignorar las diferencias entre mayúsculas y minúsculas al comparar dos cadenas, debemos utilizar **compareToIgnoreCase( )**, cuya forma es:

int compareTolgnoreCase(String *str*)

Este método devuelve los mismos resultados que **compareTo( )**, salvo que las diferencias entre mayúsculas y minúsculas se ignoran. Si se utiliza este método en el programa anterior, la palabra "Ahora" ya no saldría como primera de la lista.

Búsqueda en las Cadenas

La clase **String** proporciona dos métodos que permiten buscar un carácter o una subcadena dentro de una cadena:

- **indexOf( )** Busca la primera aparición de un carácter o subcadena.
- **lastIndexOf( )** Busca la última aparición de un carácter o subcadena.

Estos dos métodos están sobrecargados de distintas formas. En todos los casos, los métodos devuelven el índice en que se encontró el carácter o subcadena, o –1 si no se encontró.

Para buscar la primera aparición de un carácter, se utiliza:

int indexOf(int *ch*)

Para buscar la última aparición de un carácter, se utiliza:

int lastIndexOf(int *ch*)

donde *ch* es el carácter buscado.

Para buscar la primera o última aparición de una subcadena, se utiliza:

int indexOf(String *str*)
int lastIndexOf(String *str*)

donde *str* especifica la subcadena.

Se puede especificar una posición de inicio para la búsqueda utilizando las siguientes formas:

int indexOf(int *ch*, int *posInicial*)
int lastIndexOf(int *ch*, int *posInicial*)

int indexOf(String *str*, int *posInicial*)
int lastlndexOf(String *str*, int *posInicial*)

Donde *posInicial* especifica el índice de la posición donde comienza la búsqueda. Para **indexOf( )**, la búsqueda se realiza desde *posInicial* hasta el final de la cadena. Para **lastIndexOf( )**, la búsqueda se realiza desde *posInicial* hasta cero.

El siguiente ejemplo muestra el uso de varios métodos para buscar dentro de **cadenas**:

```
// Ejemplo del uso de indexOf() y lastIndexOf().
class indexOfDemo {
  public static void main(String args[]) {
    String s = "Ahora es el momento de que todos los " +
               "hombres buenos vengan a ayudar a su país.";

    System.out.println(s);
    System.out.println("indexOf(e) = " +
                       s. indexOf ( 'e' ) ) ;
    System.out.println("lastIndexOf(e) = " +
                       s.lastIndexOf('e'));
    System.out.println("indexOf(es) = " +
                       s.indexOf("es")) ;
    System.out.println("lastIndexOf(es) = " +
                       s.lastIndexOf("es"));
    System.out.println("indexOf(e, 10) = " +
                       s.indexOf('e' , 10));
    System.out.println("lastIndexOf(e, 50) = " +
                       s.lastIndexOf('e', 50));
    System.out.println("indexOf(es, 10) = " +
                       s. indexOf ("es", 10));
    System.out.println("lastIndexOf(es, 50) = " +
                       s.lastIndexOf ("es", 50));
  }
}
```

Ésta es la salida del programa:

```
Ahora es el momento de que todos los hombres buenos vengan
a ayudar a su país.
indexOf(e) = 6
lastlndexOf(e) = 53
indexOf(es) = 6
lastlndexOf(es) = 42
```

```
indexOf(e, 10) = 15
lastlndexOf(e, 50) = 47
indexOf(es, l0) = 42
lastIndexOf(es, 50) = 42
```

Modificación de una cadena

Dado que los objetos **String** son inmutables, cada vez que se quiera modificar un objeto **String** se debe o bien copiarlo en un objeto del tipo **StringBuffer** o **StringBuilder**, o bien utilizar uno de los siguientes métodos de la clase **String** los cuales construyen una nueva copia de la cadena con las modificaciones respectivas.

substring( )

Se puede extraer una subcadena utilizando el método **substring( )**. Este método tiene dos formas. La primera es:

String substring (int *posInicial*)

Donde *posInicial* especifica el índice donde comienza la subcadena. Esta forma devuelve una copia de la subcadena que comienza en *posInicial* y sigue hasta el final de la cadena que llama al método.

La segunda forma del método **substring( )** permite especificar tanto el índice de inicio como el índice final de la subcadena:

String substring (int *posInicial*, int *posFinal*)

Aquí, *posInicial* especifica el índice de inicio, y *posFinal* el punto de parada. La cadena devuelta contiene todos los caracteres desde el índice inicial hasta el índice final, pero sin incluirlo.

El siguiente programa utiliza **substring( )** para reemplazar todas las apariciones de una subcadena dentro de una cadena por otra:

```
// Reemplazo de subcadenas.
class StringReplace {
  public static void main(String args[]) {
    String org = "This is a test. This is, too.";
    String search = "is";
    String sub = "was";
    String result = "";
    int i;

    do { // reemplazar subcadenas
      System.out.println(org);
      i = org.indexOf(search);
      if(i != -1) {
        result = org.substring(0, i);
        result = result + sub;
        result = result + org.substring(i + search.length( ));
        org = result;
      }
      while(i != -1);

  }
}
```

La salida del programa se muestra a continuación:

```
This is a test. This is, too.
Thwas is a test. This is, too.
Thwas was a test. This is, too.
Thwas was a test. Thwas is, too.
Thwas was a test. Thwas was, too.
```

concat( )

Se pueden concatenar dos cadenas utilizando el método **concat( )**, como se muestra a continuación:

String concat(String *str*)

Este método crea un nuevo objeto que contiene la cadena invocante con los contenidos de *str* añadidos al final. **concat( )** hace la misma función que el operador +. Por ejemplo:

```
String s1 = "uno";
String s2 = s1.concat("dos");
```

pone la cadena "unodos" en **s2**. Esto genera el mismo resultado que la siguiente secuencia:

```
String s1 = "uno";
String s2 = s1 + "dos";
```

replace( )

El método **replace( )** tiene dos formas. La primera **reemplaza** todas las apariciones de un carácter en la cadena que invoca por otro carácter. Tiene la siguiente forma general:

String replace(char *original,* char ***reemplazo***)

Donde *original* especifica el carácter a ser ***reemplazado*** por el carácter especificado. El método devuelve la cadena resultante. Por ejemplo.

```
String s = "Hola".replace('l', 'w');
```

pone la cadena "Howa" en **s.**

La segunda forma del método **replace( )** reemplaza una secuencia de caracteres por otra. El método está definido como:

String replace(CharSequence *original,* CharSequence ***reemplazo***)

trim( )

El método **trim( )** devuelve una copia de la cadena invocante de la que se han quitado todos los espacios en blanco que pudiera tener al principio y al final. Tiene esta forma general:

String trim( )

He aquí un ejemplo:

```
String s = " Hola Mundo  ".trim();
```

pone la cadena "Hola Mundo" en **s.**

El método **trim( )** es bastante útil para procesar comandos de usuario. Por ejemplo, el siguiente programa pide al usuario su país y luego muestra la capital de ese país. El ejemplo utiliza **trim( )** para quitar los espacios en blanco iniciales o finales que el usuario haya introducido sin darse cuenta.

```
// Ejemplo del método de trim( ).
import java.io.*;

class UseTrim {
  public static void main(String args[])
    throws IOException
  {

  // crear un objeto BufferedReader con System.in
  BufferedReader br = new
    BufferedReader(new InputStreamReader(System.in));
  String str;

  System.out.println("Escriba 'fin' para terminar.");
  System.out.println("Escriba País: ");
  do {
      str = br.readLine( );
      str = str.trim( ); // quitar espacios en blanco

      if(str.equals("México"))
        System.out.println("La capital es Ciudad de México");
      else if (str .equals ("Argentina"))
        System.out.println("La capital es Buenos Aires.");
      else if(str.equals("España"))
        System.out.println("La capital es Madrid.");
      else if(str.equals("El Salvador"))
        System.out.println("La capital es San Salvador.");
      // ...
    } while(!str.equals("fin"));
  }
}
```

Conversión de datos mediante valueOf( )

El método **valueOf( )** convierte datos desde su formato interno hasta una forma legible por los humanos. Es un método estático que se sobrecarga dentro de la clase **String** para todos los tipos de Java incorporados, de modo que cada tipo se puede convertir adecuadamente en una cadena. **valueOf( )** también está sobrecargado para el tipo **Object**, por lo que un objeto de cualquier tipo de clase creado por el programador también se puede usar como argumento. Recuerde que **Object** es una superclase para todas las clases. He aquí algunas formas del método:

static String valueOf(double *num*)
static String valueOf(long *num*)
static String valueOf(Object *ob*)
static String valueOf(char *chars*[ ])

Tal como ya hemos visto, **valueOf( )** es llamado cuando se necesita una representación en forma de cadena de algún otro tipo de datos, por ejemplo en operaciones de concatenación. Se

puede llamar a este método directamente con cualquier tipo de dato y obtener una representación razonable en forma de **cadena**. Todos los tipos simples se convierten a su representación **String** común. Cualquier objeto que se le pase a **valueOf( )** devolverá el resultado de la llamada al método **toString( )** de dicho objeto. De hecho, se puede simplemente llamar a **toString( )** directamente y obtener el mismo resultado.

Para la mayoría de los arreglos, **valueOf( )** devuelve una cadena algo críptica, lo que indica que es un arreglo de algún tipo. Para arreglos de tipo **char**, sin embargo, se crea un objeto **String** que contiene los caracteres del arreglo **char**. He aquí una versión especial de **valueOf( )** que permite especificar un subconjunto de un arreglo **char**. Tiene la forma general:

static String valueOf(char *chars*[ ], int *posInicial*, int *numChars*)

Donde *chars* es el arreglo que contiene los caracteres, *posInicial* es el índice del arreglo de caracteres en que comienza la subcadena deseada, y *numChars* especifica la longitud de la subcadena.

Cambio entre mayúsculas y minúsculas dentro de una cadena

El método **toLowerCase( )** convierte todos los caracteres de una cadena de mayúsculas a minúsculas. El método **toUpperCase( )** convierte todos los caracteres de una cadena de minúsculas a mayúsculas. Los caracteres no alfabéticos, como los números, no se ven afectados. La forma general de estos métodos es:

String toLowerCase( )
String toUpperCase( )

Ambos métodos devuelven un objeto **String** que contiene el equivalente en mayúsculas o minúsculas de la **cadena** que invoca.

He aquí un ejemplo que usa **toLowerCase( )** y **toUpperCase( )**:

```
// Uso de toUpperCase () y toLowerCase () .

class ChangeCase {
  public static void main(String args[])
  {
    String s = "Esto es una prueba.";

    System.out.println("Original: " + s);

    String mayúsculas = s.toUpperCase();
    String minúsculas = s.toLowerCase();

    System.out.println("En mayúsculas: " + mayúsculas);
    System.out.println("En minúsculas: " + minúsculas);
  }
}
```

La salida producida por el programa es la siguiente:

```
Original: Esto es una prueba.
En mayúsculas: ESTO ES UNA PRUEBA.
En minúsculas: esto es una prueba.
```

Existen versiones sobrecargadas de **toLowerCase( )** y **toUpperCase( )** que permiten especificar un objeto de tipo **Locale** para controlar la conversión.

Otros métodos para trabajar con cadenas

Además de los métodos mencionados anteriormente, la clase **String** incluye otros tantos métodos. La siguiente tabla es un resumen de métodos disponibles en la clase String.

Método	Descripción
int codePointAt(int *i*)	Devuelve el punto de código Unicode en la posición especificada por *i*
int codePointBefore(int *i*)	Devuelve el punto de código Unicode en la posición que precede a *i*
int codePointCount(int *inicio*, int *fin*)	Devuelve el número de puntos de código Unicode en la porción de la cadena entre las posiciones *inicio* y *fin*–1.
boolean contains(CharSequence *str*)	Devuelve **verdadero** si el objeto que invoca contiene la cadena especificada por *str*. Devuelve **falso** en caso contrario.
boolean contentEquals(CharSequence *str*)	Devuelve **verdadero** si la cadena que realiza la invocación contiene el mismo texto que *str*. En caso contrario devuelve **falso**.
boolean contentEquals(StringBuffer *str*)	Devuelve **verdadero** si la cadena que realiza la invocación contiene el mismo texto que *str*. En caso contrario devuelve **falso**.
static String format (String *fmtstr*, Object … *args*)	Devuelve una cadena en el formato especificado por *fmtstr*. En el Capítulo 18 se habla a detalle del formato de cadenas.
static String format(Locale loc, String *fmtstr*, Object … *args*)	Devuelve una cadena en el formato especificado por *fmtstr.* El formato está dirigido por el objeto Locale. En el Capítulo 18 se habla a detalle del formato de cadenas.
boolean isEmpty( )	Devuelve **verdadero** si la cadena que realiza la invocación no contiene caracteres y tiene una longitud de cero. Este método fue añadido por Java SE 6.
boolean matches (String *regExp*)	Devuelve **verdadero** si la cadena que invoca corresponde con la expresión regular establecida en *regExp*. En caso contrario devuelve **falso**.
int offsetByCodePoints(int *start*, int *num*)	Devuelve el índice si la cadena que invoca es *num* puntos de código después del incio de índice especificado por *start*.
String replaceFirst (String *regExp*, String *newStr*)	Devuelve una cadena en la cual la primera subcadena que coincide con la expresión regular establecida en *regExp* es reemplazada por la cadena *newStr*.
String replaceAll (String *regExp*, String *newStr*)	Devuelve una cadena en la cual todas las subcadenas que coinciden con la expresión regular establecida en *regExp* son remplazadas por la cadena *newStr*.

Método	Descripción
String[ ] split (String *regExp*)	Descompone a la cadena que realiza la invocación en partes y devuelve un arreglo que contiene dicho resultado. Cada parte es delimitada por la expresión regular definida por *regExp*.
String [ ] split (String *regExp*, int *max)*	Descompone la cadena que invocó en partes y regresa un arreglo que contiene dicho resultado. Las partes son separadas acorde con lo que indique la expresión regular definida por *regExp*. El número de partes es especificado por *max*. Si *max* contiene un valor positivo, la expresión regular se aplica como máximo max–1 veces, y la última cadena en el arreglo resultante contiene el sobrante de la cadena que invoca. En caso contrario, si *max* es un valor negativo entonces la expresión regular se aplica tantas veces como sea posible y la cadena es completamente separada en partes, incluso se conservan las cadenas vacías que pudieran quedar al final del arreglo resultante. Si *max* es cero, entonces la expresión regular se aplica tantas veces como sea posible, la cadena es completamente separada en partes y si quedaran cadenas vacías insertadas al final del arreglo resultante, éstas se eliminan.
CharSequence subSequence (int *posInicial*, int *posFinal*)	Devuelve una subcadena tomada de la cadena que realizó la invocación, comenzando en *posInicial* y hasta *posFinal*. Este método es utilizado por la interfaz **CharSequence**, la cual es implementada por la clase **String**.

Observe que varios de estos métodos utilizan expresiones regulares. Las expresiones regulares se describen en el Capítulo 27.

StringBuffer

StringBuffer es una clase semejante a **String** que proporciona buena parte de la funcionalidad de las cadenas. Como sabemos, **String** representa secuencias de caracteres de inmutables de longitud fija. En contraste, **StringBuffer** representa secuencias de caracteres que pueden crecer y sobrescribirse. A un objeto **StringBuffer** se le puede insertar o añadir al final caracteres y subcadenas. **StringBuffer** crecerá automáticamente para hacer espacio para estas adiciones y, a menudo, tiene más caracteres asignados en memoria que los que realmente necesita, para dejar espacio para crecer en tamaño. Java utiliza ambas clases intensivamente, pero muchos programadores sólo manejan **String** y dejan a Java manipular **StringBuffer** de manera automática utilizando el operador sobrecargado **+**.

Constructores StringBuffer

StringBuffer define los siguientes cuatro constructores:

StringBuffer( )
StringBuffer(int *tamaño*)

StringBuffer(String *str*)
StringBuffer(CharSequence *chars*)

El constructor por omisión (el que no lleva parámetros) reserva espacio para 16 caracteres sin reasignación de memoria. La segunda versión acepta un argumento entero que explícitamente fija el tamaño del espacio reservado. La tercera versión acepta como argumento un objeto **String** que fija los contenidos iniciales del objeto **StringBuffer** y reserva espacio para 16 caracteres más sin reasignación. **StringBuffer** asigna espacio para 16 caracteres adicionales cuando no se solicita una longitud explícita, debido a que la reasignación es un proceso costoso en tiempo. Además que reasignaciones frecuentes pueden fragmentar la memoria. Asignando espacio para unos pocos caracteres adicionales, **StringBuffer** reduce el número de reasignaciones que puedan surgir. El cuarto constructor crea un objeto que contiene la secuencia de caracteres definida por *chars*.

length( ) y capacity( )

La longitud actual de un **StringBuffer** se puede obtener por medio del método **length( )**, mientras que la capacidad total asignada se obtiene con el método **capacity( )**. Sus formas generales son:

int length( )
int capacity( )

He aquí un ejemplo:

```
// Longitud y capacidad de un StringBuffer.
class StringBufferDemo {
  public static void main(String args[]) {
    StringBuffer sb = new StringBuffer("Hola");

    System.out.println("valor = " + sb);
    System.out.println("longitud = " + sb.length());
    System.out.println("capacidad = " + sb.capacity());
  }
}
```

La salida del programa muestra cómo **StringBuffer** reserva espacio extra para manipulaciones adicionales:

```
valor = Hola
longitud = 4
capacidad = 20
```

La variable **sb** se inicializa con la cadena "Hola", su longitud es 4 y su capacidad es 20 debido a que se añade automáticamente espacio para otros 16 caracteres.

ensureCapacity( )

Si se desea preasignar espacio para un cierto número de caracteres después de que se ha construido un **StringBuffer**, se puede utilizar **ensureCapacity( )**. Este método es muy útil cuando se conoce de antemano que se van a añadir muchas cadenas pequeñas a un **StringBuffer**. **ensureCapacity( )** cuya forma general de es:

void ensureCapacity(int *capacidad*)

Donde *capacidad* especifica el tamaño del espacio en donde se almacenará la información.

setLength( )

Para fijar la longitud del espacio de almacenamiento dentro de un objeto **StringBuffer**, se utiliza el método **setLength( )**. Su forma general es:

void setLength(int *len*)

Donde *len* especifica la longitud del búfer. Este valor no debe ser negativo.

Al aumentar el tamaño del espacio de almacenamiento, se rellena la cadena con caracteres nulos al final de la misma. Si se llama a **setLength( )** con un valor menor que el actualmente devuelto por **length( )**, entonces los caracteres almacenados más allá de la nueva longitud se perderán. El programa ejemplo **setCharAtDemo** en la siguiente sección utiliza **setLength( )** para acortar un **StringBuffer**.

charAt( ) y setCharAt( )

El valor de un carácter específico en un **StringBuffer** se puede obtener por medio del método **charAt( )**. También se puede asignar un valor a un carácter dentro del **StringBuffer** utilizando el método **setCharAt( )**. Sus formas generales son:

char charAt(int *donde*)
void setCharAt(int *donde*, char *ch*)

Para **charAt( )**, *donde* especifica el índice del carácter que se desea obtener. Para **setCharAt( )**, *donde* especifica el índice del carácter cuyo valor será alterado, y *ch* es el nuevo valor de dicho carácter. Para ambos métodos, *donde* no debe ser negativo y no debe especificar una posición más allá del tamaño del espacio de almacenamiento.

El siguiente ejemplo muestra el uso de **charAt( )** y **setCharAt( )**:

```
// Uso de charAt () y setCharAt ().
class setCharAtDemo {
  public static void main{String args[]) {
    StringBuffer sb = new StringBuffer{"Hola.");
    System.out.println{"StringBuffer antes = " + sb);
    System.out.println("charAt(l) antes =" + sb.charAt{l));
      sb.setCharAt{l, 'i');
      sb.setLength(2) ;
      System.out.println{"StringBuffer después = " + sb);
      System.out.println("charAt(l) después = " + sb.charAt(l));
  }
}
```

La salida generada por este programa es la siguiente:

```
StringBuffer antes = Hola.
charAt(l) antes = o
StringBuffer después = Hi
charAt(l) después = i
```

getChars( )

Para copiar una subcadena de un objeto **StringBuffer** dentro de un arreglo, se utiliza el método **getChars( )**. Este método tiene la siguiente forma general:

void getChars(int *inicioOrigen*, int *finalOrigen*, char *destino*[ ], int *inicioDestino*)

Aquí, *inicioOrigen* es el índice donde comienza la subcadena, y *finalOrigen* es un índice posterior en una unidad al del final de la subcadena deseada.

Esto significa que la subcadena contiene los caracteres desde *inicioOrigen* hasta *finalOrigen*-1. El arreglo que recibe los caracteres se especifica en *destino*. El índice dentro de *destino* a partir del cual se copia la subcadena se proporciona en *inicioDestino*. Hay que tener cuidado en asegurar que el arreglo destino sea lo suficientemente grande como para albergar todos los caracteres de la subcadena especificada.

append( )

El método **append( )** concatena la representación textual de cualquier tipo de datos al final del objeto **StringBuffer** que llama al método. Este método tiene varias versiones sobrecargadas. He aquí algunas de sus formas:

StringBuffer append(String *str*)
StringBuffer append(int *num*)
StringBuffer append(Object *obj*)

Se llama a **String.valueOf( )** por cada parámetro para obtener su representación como cadena. El resultado se añade al objeto **StringBuffer**. La cadena resultante es devuelta por cada versión de **append( )**. Esto permite encadenar llamadas sucesivas unas tras otras, como se muestra en el siguiente ejemplo:

```
// Ejemplo con append().
class appendDemo {
  public static void main(String args[]) {
    String s;
    int a = 42;
    StringBuffer sb = new StringBuffer(40);

    s = sb.append("a = ") .append(a) .append("!") .toString( );
    System.out.println(s);
  }
}
```

La salida de este ejemplo se muestra a continuación:

```
a = 42!
```

Cuando más a menudo se llama al método **append( )** es al utilizar el operador **+** sobre objetos **String**. Java cambia automáticamente las modificaciones de una instancia **String** en operaciones similares sobre una instancia **StringBuffer**. Así que una concatenación llama a **append( )** sobre un objeto **StringBuffer**. Después de que la concatenación se ha llevado a cabo, el compilador introduce una llamada a **toString( )** para transformar el **StringBuffer** modificable en un **String** constante. Toda esta complicación puede parecer poco razonable. ¿Por qué no tener simplemente una clase de cadena y que se comporte más o menos como **StringBuffer**? La respuesta está en el rendimiento. Hay muchas optimizaciones que el intérprete de Java puede hacer si sabe que los objetos **String** son inmutables. Afortunadamente, Java esconde la mayor parte de la complejidad de la conversión entre **String** y **StringBuffer**. De hecho, muchos programadores nunca sentirán la necesidad de usar **StringBuffer** directamente, y serán capaces de expresar la mayoría de las operaciones en términos del operador **+** sobre variables **String**.

insert( )

El método **insert( )** inserta una cadena dentro de otra. Está sobrecargado para aceptar valores de todos los tipos simples, además de instancias de tipo **String y Object.** Al igual que **append( )**, llama a **String.valueOf( )** para obtener la representación como cadena del valor con el que es llamado. Esta cadena se inserta entonces en el objeto **StringBuffer** que invoca. Éstas son algunas de sus formas:

StringBuffer insert(int *indice,* String *str*)
StringBuffer insert(int *indice,* char *ch*)
StringBuffer insert(int *indice,* Object *obj*)

Donde, *indice* especifica el índice dentro del objeto **StringBuffer** en cuyo punto se inserta la cadena.

El siguiente programa ejemplo inserta "trabajo con "entre "Yo" y "Java":

```
// Ejemplo con insert().
class insertDemo {
  public static void main(String args[]) {
    StringBuffer sb =new StringBuffer(" ¡Yo Java!");

    sb.insert(4, "trabajo con ");
    System.out.println(sb) ;
  }
}
```

La salida del programa es ésta:

```
¡Yo trabajo con Java!
```

reverse( )

Se puede invertir el orden de los caracteres en un objeto **StringBuffer** utilizando el método **reverse( )**, mostrado a continuación:

StringBuffer reverse( )

Este método devuelve el objeto sobre el que fue llamado con el orden invertido. El siguiente programa muestra el uso de **reverse( )**:

```
// Utilizando de reverse( ) para invertir un StringBuffer.
class ReverseDemo {
  public static void main(String args[]) {
    StringBuffer s = new StringBuffer("abcdef");

    System.out.println(s);
    s.reverse () ;
    System.out.println(s);
  }
}
```

Ésta es la salida producida por el programa:

```
abcdef
fedcba
```

PARTE II

delete( ) y deleteCharAt( )

Se pueden eliminar caracteres de un **StringBuffer** por medio de los métodos **delete( )** y **deleteCharAt( )**. Estos métodos se muestran aquí:

StringBuffer delete(int *posInicial*, int *posFinal*)
StringBuffer deleteCharAt(int *pos*)

El método **delete( )** borra una sucesión de caracteres del objeto que lo invoca. Aquí, *posInicial* especifica el índice del primer carácter que se ha de borrar, y *posFinal* es superior en una unidad al del último carácter que se ha de borrar. Es decir, la subcadena borrada va desde *posInicial* hasta *posFinal*-l. El método **delete( )** devuelve el objeto StringBuffer resultante.

El método **deleteCharAt( )** borra el carácter en la posición especificada por *pos*, devolviendo el objeto **StringBuffer** resultante.

Veamos un programa que ejemplifica el uso de los métodos **delete( )** y **deleteCharAt( )**:

```
// Ejemplo con delete()y deleteCharAt()
class deleteDemo {
  public static void main(String args[]) {
    StringBuffer sb = new StringBuffer("Esto es una prueba.");

    sb.delete(4, 7);
    System.out.println("Después de delete: " + sb);

    sb.deleteCharAt(0);
    System.out.println("Después de deleteCharAt: " + sb);
  }
}
```

Se produce la siguiente salida:

```
Después de delete: Esto una prueba.
Después de deleteCharAt: sto una prueba.
```

replace( )

En un **StringBuffer** es posible reemplazar un conjunto de caracteres por otro utilizando el método **replace( )**. Su firma se muestra a continuación:

StringBuffer replace(int *posInicial*, int *posFinal*, String *str*)

La subcadena que se reemplaza viene especificada por los índices *posInicial* y *posFinal*. Así, la subcadena desde *posInicial* hasta *posFinal*-1 es reemplazada. La cadena reemplazante se pasa en *str*. El método devuelve el objeto **StringBuffer** resultante.

El siguiente programa muestra el uso de **replace( )**:

```
// Ejemplo con replace()
class replaceDemo {
  public static void main(String args()) {
    StringBuffer sb = new StringBuffer("Esto es una prueba.");

    sb.replace(5, 7, "era");
    System.out.println("Después de replace: " + sb);
  }
}
```

y la salida es:

```
Después de replace: Esto era una prueba.
```

substring( )

Es posible obtener una porción de un **StringBuffer** mediante el método **substring( )**, el cual tiene las siguientes dos formas:

String substring(int *posInicial*)
String substring(int *posInicial*, int *posFinal*)

La primera forma devuelve la subcadena que empieza en *posInicial* y sigue hasta el final del objeto **StringBuffer** que invoca. La segunda forma devuelve la subcadena que empieza en *posInicial* y termina en *posFinal*-l. Estos métodos funcionan igual que los definidos para **String**, ya descritos anteriormente.

Otros métodos para trabajar con StringBuffer

Adicionalmente a los métodos descritos, la clase **StringBuffer** incluye muchos otros métodos. La siguiente tabla muestra algunos más.

Método	Descripción
StringBuffer appendCodePoint(int *ch*)	Agrega un punto de código Unicode al final del objeto invocante. El método devuelve una referencia al objeto.
int codePointAt(int *i*)	Devuelve el punto de código Unicode en la posición especificada por *i*.
int codePointBefore(int *i*)	Devuelve el punto de código Unicode en la posición anterior a la especificada por *i*.
int codePointCount(int *inicio*, int *fin*)	Devuelve el número de puntos de código Unicode en la porción de la cadena que invoca que se encuentran entre *inicio* y *fin* – 1.
int indexOf(String *str*)	Busca en la cadena que invoca la primera ocurrencia de *str*. Devuelve el índice de la coincidencia o –1 si no se encuentra ninguna coincidencia.
int indexOf(String *str*, int *inicio*)	Busca en la cadena que invoca la primera ocurrencia de *str*, a partir de la posición *inicio*. Devuelve el índice de la coincidencia o –1 si no se encuentra ninguna coincidencia.
int lastIndexOf(String *str*)	Busca en la cadena que invoca la última ocurrencia de *str*. Devuelve el índice de la coincidencia o –1 si no se encuentra ninguna coincidencia.
int lastIntexOf(String *str*, int *inicio*)	Busca en la cadena que invoca la última ocurrencia de *str*, a partir de la posición *inicio*. Devuelve el índice de la coincidencia o –1 si no se encuentra ninguna coincidencia.
int offsetByCodePoints(int *inicio*, int *n*)	Devuelve el índice en la cadena invocante que esta *n* puntos de código Unicode más allá del índice inicial especificado por *inicio*.

Método	Descripción
CharSequence subsequence (int *inicio*, int *fin*)	Devuelve una subcadena de la cadena que invoca, comenzando en *inicio* y que concluye en la posición *fin*. Este método es utilizado por la interfaz **CharSequence**, la cual es implementada en la clase **StringBuffer**.
void trimToSize( )	Reduce el tamaño del espacio de almacenamiento de caracteres del objeto que invoca para ajustarlo exactamente al contenido actual.

El siguiente programa muestra el uso de los métodos **indexOf( )** y **lastIndexOf( )**:

```
class IndexOfDemo {
  public static void main(String args[]) {
    StringBuffer sb = new StringBuffer("uno dos uno");
    int i;

    i.sb.indexOf("uno");
    System.out.println("Primer índice: " + i);

    i = sb.lastIndexOf("uno");
    System.out.println("Último índice: " + i);
  }
}
```

Ésta es la salida mostrada por el programa:

```
Primer índice: 0
Último índice: 8
```

StringBuilder

A partir del JDK 5 se anexa la clase **StringBuilder** a las capacidades de gestión de cadenas de Java. La clase **StringBuilder** es idéntica a la clase **StringBuffer** excepto por una cosa: no es una clase sincronizada, lo cual significa que no es seguro utilizarla en programas que trabajan con múltiples hilos. La ventaja de **StringBuffer** es el aumento de rendimiento en términos de velocidad. Sin embargo, en los casos en los que se trabaja con programación multihilo debemos utilizar **StringBuffer** en lugar de **StringBuilder**.

CAPÍTULO 16

Explorando java.lang

Este capítulo trata sobre las clases e interfaces definidas en **java.lang**. Como sabemos, **java.lang** se importa automáticamente en todos los programas. El paquete **java.lang** contiene las clases e interfaces fundamentales para prácticamente cualquier programa en Java. Es el paquete de Java más ampliamente utilizado.

java.lang incluye las siguientes clases:

Boolean	InheritableThreadLocal	Runtime	System
Byte	Integer	RuntimePermission	Thread
Character	Long	SecurityManager	ThreadGroup
Class	Math	Short	ThreadLocal
ClassLoader	Number	StackTraceElement	Throwable
Compiler	Object	StrictMath	Void
Double	Package	String	
Enum	Process	StringBuffer	
Float	ProcessBuilder	StringBuilder	

También hay dos clases definidas por **Character**: **Chacter.Subset** y **Charter.UnicodeBlock**.

java.lang también define las siguientes interfaces:

Appendable	Comparable	Runnable
CharSequence	Iterable	
Cloneable	Readable	

Muchas de las clases contenidas en **java.lang** contienen métodos catalogados como en desuso, la mayoría de los cuales aparecieron en Java 1.0. Estos métodos en desuso son aún provistos por Java para soportar cualquier legado de código y no se recomiendan para código nuevo. La mayoría de los desusos tuvieron lugar antes de Java SE 6, y estos métodos en desuso no se discuten aquí.

Envoltura de tipos primitivos

Como se mencionó en la primera parte de este libro, Java utiliza tipos primitivos como **int** y **char**, por razones de rendimiento. Estos tipos de datos no son parte de la jerarquía de objetos. Éstos se pasan por valor al método y no pueden ser pasados directamente por referencia. Tampoco existe alguna forma en que dos métodos hagan referencia a la *misma instancia* de un **int**. En ocasiones, se requiere la creación de un objeto para representar uno de estos tipos primitivos. Por ejemplo, existe una colección de clases que se discutirá en el Capítulo 17 que trabaja sólo con objetos; para almacenar un tipo primitivo en una de esas clases, se necesita envolver al tipo primitivo en una clase. Para solucionar esta necesidad Java provee clases que corresponden a cada tipo primitivo. En esencia, estas clases encapsulan, o *envuelven*, los tipos primitivos dentro de una clase. De este modo, estas clases son comúnmente referidas como tipos *envueltos*. Los tipos envueltos fueron introducidos en el Capítulo 12. Y se examinan a detalle aquí.

Number

La clase abstracta **Number** define una superclase que está implementada por las clases que envuelven los tipos numéricos **byte**, **short**, **int**, **long**, **float**, y **double**. La clase **Number** tiene métodos abstractos que devuelven el valor del objeto en cada uno de los diferentes formatos. Por ejemplo, **doubleValue( )** devuelve el valor como **double**, **floatValue( )** devuelve el valor como **float**, y así sucesivamente. Estos métodos son los siguientes:

```
byte byteValue( )
double doubleValue( )
float floatValue( )
int intValue( )
long longValue( )
short shortValue( )
```

Los valores devueltos por estos métodos pueden estar redondeados.

Number tiene seis subclases concretas que contienen valores explícitos de cada tipo numérico: **Double**, **Float**, **Byte**, **Short**, **Integer** y **Long**.

Double y Float

Double y **Float** son envoltorios para valores de punto flotante del tipo **double** y **float** respectivamente. Los constructores para **Float** son éstos:

```
Float(double num)
Float(float num)
Float(String str) throws NumberFormatException
```

Como se ve, los objetos **Float** se pueden construir con valores de los tipos **float** o **double**. También se pueden construir a partir de la representación como cadena de un número de punto flotante.

Los constructores de **Double** son los siguientes:

```
Double(double num)
Double(String str) throws NumberFormatException
```

Los objetos **Double** se pueden construir con un valor **double** o una cadena que contenga el valor de punto flotante.

Los métodos definidos por **Float** se muestran en la Tabla 16-1. Los métodos definidos por **Double** se muestran en la Tabla 16-2. Tanto **Float** como **Double** definen las siguientes constantes:

MAX_EXPONENT	Máximo exponente (agregado por Java SE 6)
MAX_VALUE	Máximo valor positivo
MIN_EXPONENT	Mínimo exponente (agregado por Java SE 6)
MIN_NORMAL	Mínimo valor normal positivo (agregado por Java SE 6)
MIN_VALUE	Mínimo valor positivo
NaN	No es un número
POSITIVE_INFINITY	Más infinito
NEGATIVE_INFINITY	Menos infinito
SIZE	El tamaño en bits del valor envuelto
TYPE	El objeto **Class** para **float** o **double**

Método	Descripción
byte byteValue( )	Devuelve el valor del objeto que invoca como un **byte**.
static int compare(float *num1*, float *num2*)	Compara el valor de *num1* y *num2*. Devuelve 0 si el valor es igual. Devuelve un valor negativo si *num1* es menor que *num2*. Devuelve un valor positivo si *num1* es mayor que *num2*.
int compareTo(Float *f*)	Compara numéricamente el valor del objeto que invoca con el valor *f*. Devuelve 0 si los valores son iguales. Devuelve un valor negativo si el objeto que invoca tiene un valor menor. Devuelve un valor positivo si el objeto que invoca tiene un valor mayor.
double doubleValue( )	Devuelve el valor del objeto que invoca como **double**.
boolean equals(Object ObjFloat)	Devuelve **verdadero** si el objeto **Float** que invoca es equivalente a ObjFloat. De lo contrario, devuelve **falso**.
static int floatToIntBits(float *num*)	Devuelve el patrón de bits de precisión simple compatible-IEEE correspondiente a *num*.
static int floatToRawIntBits(float *num*)	Devuelve el patrón de bits de precisión simple compatible-IEEE correspondiente a *num*. El valor Nan se conserva.
float floatValue( )	Devuelve el valor del objeto que invoca como **float**.
int hashCode( )	Devuelve el código de dispersión del objeto que invoca.
static float intBitsToFloat(int *num*)	Devuelve el equivalente **float** del patrón de bits de precisión simple compatible-IEEE especificado por *num*.
int intValue( )	Devuelve el valor del objeto que invoca como **int**.
boolean isInfinite( )	Devuelve **verdadero** si el objeto que invoca contiene un valor infinito. En caso contrario, devuelve **falso**.
static boolean isInfinite(float *num*)	Devuelve **verdadero** si *num* especifica un valor infinito. De lo contrario devuelve **falso**.

Tabla 16-1 Los métodos definidos por la clase **Float**

Método	Descripción
boolean isNaN( )	Devuelve **verdadero** si el objeto que invoca contiene un valor que no es un número. De lo contrario, devuelve **falso**.
static boolean isNaN(float *num*)	Devuelve **verdadero** si *num* especifica un valor que no es un número. De lo contrario devuelve **falso**.
long longValue( )	Devuelve el valor del objeto que invoca como **long**.
static float parseFloat(String *str*) throws NumberFormatException	Devuelve el equivalente **float** del número contenido en la cadena especificada por *str* utilizando la base 10.
short shortValue( )	Devuelve el valor del objeto que invoca como **short**.
static String toHexString(float *num*)	Devuelve una cadena que contiene el valor de *num* en formato hexadecimal
String toString( )	Devuelve la cadena equivalente del objeto que invoca.
static String toString(float *num*)	Devuelve la cadena equivalente del valor especificado por *num*.
static Float valueOf(float *num*)	Devuelve un objeto **Float** que contiene el valor especificado por *num*.
static Float valueOf(String *str*) throws NumberFormatException	Devuelve el objeto **Float** que contiene el valor especificado por la cadena *str*.

TABLA 16-1 Los métodos definidos por la clase **Float** (*continuación*)

Método	Descripción
byte byteValue( )	Devuelve el valor del objeto que invoca como un **byte**.
static int compare(double *num1*, double *num2*)	Compara los valores de *num1* y *num2*. Devuelve 0 si el valor es igual. Devuelve un valor negativo si *num2* es menor a *num2*. Devuelve un valor positivo si *num1* es mayor que *num2*.
int compareTo(Double *d*)	Compara el valor numérico del objeto que invoca con el de *d*. Devuelve 0 si los valores son iguales. Devuelve un valor negativo si el objeto que invoca es menor a *d*. Devuelve un valor positivo si el objeto que invoca es mayor que *d*.
static long doubleToLongBits(double *num*)	Devuelve el patrón de bits de doble precisión compatible IEEE que corresponde al *num*.
static long doubleToRawLongBits(double *num*)	Devuelve el patrón de bits de doble precisión compatible IEEE que corresponde al *num*. El valor NaN se conserva.
double doubleValue( )	Devuelve el valor del objeto que invoca como un *double*.
boolean equals(Object *ObjDouble*)	Devuelve **verdadero** si el objeto **Double** que invoca es equivalente a *ObjDouble*. De lo contrario, devuelve **falso**.
float floatValue( )	Devuelve el valor del objeto que invoca como un **float**.
int hashcode( )	Devuelve el código de dispersión del objeto que invoca.
int intValue( )	Devuelve el valor del objeto que invoca como un **int**.

TABLA 16-2 Los métodos definidos por la clase **Double**

Método	Descripción
boolean isInfinite( )	Devuelve **verdadero** si el objeto que invoca contiene un valor infinito. De lo contrario, devuelve **falso**.
static boolean isInfinite(double *num*)	Devuelve **verdadero** si *num* especifica un valor infinito. De lo contrario devuelve **falso**.
boolean isNaN( )	Devuelve **verdadero** si el objeto que invoca contiene un valor que no es un *número*. De lo contrario, devuelve **falso**.
static boolean isNaN(double *num*)	Devuelve **verdadero** si *num* especifica un valor que no es un número. De lo contrario, devuelve **falso**.
static double longBitsToDouble(long *num*)	Devuelve el equivalente **double** del patrón de bits de doble precisión IEEE compatible especificado por *num*.
long longValue( )	Devuelve el valor del objeto que invoca como un **long**.
static double parseDouble(String *str*) throws NumberFormatException	Devuelve el equivalente **double** del número contenido en la cadena especificada por *str* utilizando base 10.
short shortValue( )	Devuelve el valor del objeto que invoca como un **short**.
static String toHexString(double *num*)	Devuelve una cadena que contiene el valor de *num* en formato hexadecimal.
String toString( )	Devuelve la cadena equivalente del objeto que invoca.
static String toString(double *num*)	Devuelve la cadena equivalente del valor especificado por *num*.
static Double valueOf(double *num*)	Devuelve un objeto **Double** que contiene el valor especificado por *num*.
static Double valueOf(String *str*) throws NumberFormatException	Devuelve un objeto **Double** que contiene el valor especificado por la cadena *str*.

TABLA 16-2 Los métodos definidos por la clase **Double** (*continuación*)

El siguiente ejemplo crea dos objetos **Double,** uno utilizando un valor **double** y el otro pasando una cadena que contiene la representación de un **double**:

```
class DoubleDemo {
  public static void main(String args[]) {
    Double d1 = new Double(3.14159);
    Double d2 = new Double("314159E-5");

    System.out.println(dl + " = " + d2 + " -> " + dl.equals(d2));
  }
}
```

Como se puede ver en la salida del programa, ambos constructores han creado instancias **Double** idénticas, por ello el método **equals( )** devuelve **verdadero**:

```
3.14159 = 3.14159 -> true
```

Los métodos isInfinite( ) e isNaN( )

Float y **Double** proporcionan los métodos **isInfinite( )** e **isNaN( )**, que ayudan a manipular dos valores **double** y **float** especiales. Estos métodos funcionan con dos valores únicos definidos por la especificación de punto flotante de IEEE: infinito y NaN (Not a Number). El método **isInfinite( )** devuelve verdadero si el valor probado es infinitamente grande o pequeño en magnitud. **isNaN( )** devuelve verdadero si el valor que se prueba no es un número.

El siguiente ejemplo crea dos objetos **Double**; uno es infinito, y el otro no es un número:

```
// Ejemplo con isInfinite() e isNaN()
class InfNaN {
  public static void main(String args[]) {
    Double d1 = new Double(1/0.);
    Double d2 = new Double(0/0.);

    System.out.println(dl + ": " + dl.isInfinite() + ", " + dl.isNaN());
    System.out.println(d2 + ": " + d2.islnfinite() + ", " + d2.isNaN());
  }
}
```

El programa genera esta salida:

```
Infinity: true, false
NaN: false, true
```

Byte, Short, Integer y Long

Las clases **Byte**, **Short**, **Integer** y **Long** son envoltorios para los tipos enteros **byte**, **short**, **int** y **long** respectivamente. Sus constructores son éstos:

Byte(byte *num*)
Byte(String str) throws NumberFormatException

Short(short *num*)
Short(String str) throws NumberFormatException

Integer(int *num*)
Integer(String str) throws NumberFormatException

Long(long *num)*
Long(String str) throws NumberFormatException

Como se ve, estos objetos se pueden construir a partir de valores numéricos o de cadenas que contengan valores válidos de números enteros.

Los métodos definidos por estas clases se muestran en las Tablas 16-3 a 16-6. Como puede observarse, estas clases definen métodos para obtener enteros a partir de cadenas o convertir cadenas de nuevo en enteros. Existen variantes de estos métodos que permiten especificar la base numérica para la conversión. Bases comunes son: 2 para binario, 8 para octal, 10 para decimal y 16 para hexadecimal.

Se definen las siguientes constantes:

MIN_ VALUE	Valor mínimo
MAX_ VALUE	Valor máximo
SIZE	El tamaño en bits de un valor envuelto
TYPE	El objeto **Class** para **byte**, **short**, **int**, o **long**

Método	Descripción
byte byteValue( )	Devuelve el valor del objeto que invoca como un **byte**.
int compareTo(Byte *b*)	Compara el valor numérico del objeto que invoca con el de *b*. Devuelve 0 si los valores son iguales. Devuelve un valor negativo si el objeto que invoca tiene menor valor. Devuelve un valor positivo si el objeto que invoca tiene mayor valor.
static Byte decode(String *str*) throws NumberFormatException	Devuelve un objeto **Byte** que contiene el valor especificado por la cadena *str*.
double doubleValue( )	Devuelve el valor del objeto que invoca como un **double**.
boolean equals(Object *ObjByte*)	Devuelve **verdadero** si el objeto **Byte** que invoca es equivalente a *ObjByte*. De lo contrario, devuelve **falso**.
float floatValue( )	Devuelve el valor del objeto que invoca como un **float**.
int hashCode( )	Devuelve el código de dispersión del objeto que invoca.
int intValue( )	Devuelve el valor del objeto que invoca como un **int**.
long longValue( )	Devuelve el valor del objeto que invoca como un **long**.
static byte parseByte(String *str*) throws NumberFormatException	Devuelve el **byte** equivalente del número contenido en la cadena especificada en *str* utilizando la base 10.
static byte parseByte(String *str*, int *base*) throws NumberFormatException	Devuelve el equivalente **byte** del número contenido en la cadena especificada en *str* usando la base especificada por base.
short shortValue( )	Devuelve el valor del objeto que invoca como un **short**.
String toString( )	Devuelve una cadena que contiene el equivalente decimal del objeto que invoca.
static String toString(byte *num*)	Devuelve una cadena que contiene el equivalente decimal de *num*.
static Byte valueOf(byte *num*)	Devuelve un objeto **Byte** que contiene el valor especificado en *num*.
static Byte valueOf(String *str*) throws NumberFormatException	Devuelve un objeto **Byte** que contiene el valor especificado por la cadena *str*.
static Byte valueOf(String *str*, int *base*) throws NumberFormatException	Devuelve un objeto **Byte** que contiene el valor especificado por la cadena *str* usando la *base* especificada.

TABLA 16-3 Los métodos definidos por **Byte**

Método	Descripción
byte byteValue( )	Devuelve el valor del objeto que invoca como un **byte**.
int compareTo(Short s)	Compara el valor numérico del objeto que invoca con el de s. Devuelve 0 si los valores son iguales. Devuelve un valor negativo si el objeto que invoca tiene menor valor. Devuelve un valor positivo si el objeto que invoca tiene mayor valor.
static Short decode(String *str*) throws NumberFormatException	Devuelve un objeto **Short** que contiene el valor especificado por la cadena *str*.
double doubleValue( )	Devuelve el valor del objeto que invoca como un **double**.
boolean equals(Object *ObjShort*)	Devuelve **verdadero** si el objeto **Short** que invoca es equivalente a *ObjShort*. De lo contrario, devuelve **falso**.
float floatValue( )	Devuelve el valor del objeto que invoca como un **float**.
int hashCode( )	Devuelve el código de dispersión del objeto que invoca.
int intValue( )	Devuelve el valor del objeto que invoca como un **int**.
long longValue( )	Devuelve el valor del objeto que invoca como un **long**.
static short parseShort(String *str*) throws NumberFormatException	Devuelve el equivalente **short** del número contenido en la cadena especificada en *str* usando base 10.
static short parseShort(String str, int *base*) throws NumberFormatException	Devuelve el equivalente **short** del número contenido en la cadena especificada en *str* usando la base especificada.
static short reverseBytes(short *num*)	Intercambia el orden los bytes más altos y los más bajos de *num* y devuelve el resultado.
short shortValue( )	Devuelve el valor del objeto que invoca como un **short**.
String toString( )	Devuelve una cadena que contiene el equivalente decimal del objeto que invoca.
static String toString(short *num*)	Devuelve una cadena que contiene el equivalente decimal de *num*.
static Short valueOf(short *num*)	Devuelve un objeto **Short** que contiene el valor especificado por *num*.
static Short valueOf(String *str*) throws NumberFormatException	Devuelve un objeto **Short** que contiene el valor especificado por la cadena *str* usando base 10.
static Short valueOf(String *str*, int *base*) throws NumberFormatException	Devuelve un objeto **Short** que contiene el valor especificado por la cadena *str* usando la *base* especificada.

TABLA 16-4 Los métodos definidos por la clase **Short**

Método	Descripción
static int bitCount(int *num*)	Devuelve el número de bits determinados en *num*
byte byteValue( )	Devuelve el valor del objeto que invoca como un **byte**.
int compareTo(Integer *i*)	Compara el valor numérico del objeto que invoca con el de *i*. Devuelve 0 si los valores son iguales. Devuelve un valor negativo si el objeto que invoca tiene menor valor. Devuelve un valor positivo si el objeto que invoca tiene mayor valor.
static Integer decode(String *str*) throws NumberFormatException	Devuelve un objeto **Integer** que contiene el valor especificado por la cadena *str*.
double doubleValue( )	Devuelve el valor del objeto que invoca como un **double**.
boolean equals(Object *ObjInteger*)	Devuelve **verdadero** si el objeto **Integer** que invoca es equivalente a *ObjInteger*. De lo contrario, devuelve **falso**.
float floatValue( )	Devuelve el valor del objeto que invoca como un **float**.
static Integer getInteger(String *nomPropiedad*)	Devuelve el valor asociado a la propiedad de entorno especificada por *nomPropiedad*. En caso de fallo, se devuelve **null**.
static Integer getInteger(String *nomPropiedad*, int *omisión*)	Devuelve el valor asociado a la propiedad de entorno especificada por *nomPropiedad*. En caso de fallo, se devuelve el valor *omisión*.
static Integer getInteger(String *nomPropiedad*, Integer *omisión*)	Devuelve el valor asociado a la propiedad de entorno especificada por *nomPropiedad*. En caso de fallo, se devuelve el valor por *omisión*.
int hashCode( )	Devuelve el código de dispersión del objeto que invoca.
static int highestOneBit(int *num*)	Determina la posición del bit de mayor orden con valor en *num*. Devuelve un valor en el cual sólo este bit está definido. Si no hay algún bit definido, entonces se devuelve cero.
int intValue( )	Devuelve el valor del objeto que invoca como un **int**.
long longValue( )	Devuelve el valor del objeto que invoca como un **long**.
static int lowestOneBit(int *num*)	Determina la posición del bit de orden menor definido en *num*. Devuelve el valor en el cual sólo este bit está definido. Si no hay algún bit definido, entonces se devuelve cero
static int numberOfLeadingZeros(int *num*)	Devuelve el número de bits de mayor orden en cero que preceden al primer bit de mayor orden definido en *num*. Si *num* es cero, se devuelve 32.

TABLA 16-5 Los métodos definidos por la clase **Integer**

Método	Descripción
static int numberOfTrailingZeros(int *num*)	Devuelve el número de bits de menor orden en cero que preceden al primer bit de menor orden definido en *num*. Si *num* es cero, se devuelve 32.
static int parseInt(String *str*) throws NumberFormatException	Devuelve el entero equivalente del número contenido en la cadena especificada en *str* utilizando la base 10.
static int parseInt(String *str*, int *base*) throws NumberFormatException	Devuelve el entero equivalente del número contenido en la cadena especificada en *str* utilizando la *base* especificada.
static int reverse(int *num*)	Invierte el orden de los bits en *num* y devuelve el resultado.
static int reverseBytes(int *num*)	Invierte el orden de los bytes en *num* y devuelve el resultado.
static int rotateLeft(int *num*, int *n*)	Devuelve el resultado de rotar *num*, *n* posiciones a la izquierda.
static int rotateRight(int *num*, int *n*)	Devuelve el resultado de rotar *num*, *n* posiciones a la derecha.
static int signum(int *num*)	Devuelve –1 si *num* es negativo, 0 si es cero y 1 si es positivo.
short shortValue( )	Devuelve el valor del objeto que invoca como un **short**.
static String toBinaryString(int *num*)	Devuelve una cadena que contiene el binario equivalente de *num*.
static String toHexString(int *num*)	Devuelve una cadena que contiene el hexadecimal equivalente de *num*.
static String toOctalString(int *num*)	Devuelve una cadena que contiene el octal equivalente de *num*.
String toString( )	Devuelve una cadena que contiene el decimal equivalente del objeto que invoca.
static String toString(int *num*)	Devuelve una cadena que contiene el decimal equivalente de *num*.
static String toString(int *num*, int *base*)	Devuelve una cadena que contiene el decimal equivalente de *num* utilizando la *base* especificada.
static Integer valueOf(int *num*)	Devuelve un objeto **Integer** que contiene el valor especificado por *num*.
static Integer valueOf(String *str*) throws NumberFormatException	Devuelve un objeto **Integer** que contiene el valor especificado por la cadena *str*.
static Integer valueOf(String *str*, int *base*) throws NumberFormatException	Devuelve un objeto **Integer** que contiene el valor especificado por la cadena *str* utilizando la *base* especificada.

TABLA 16-5 Los métodos definidos por la clase **Integer** (*continuación*)

Método	Descripción
static int bitCount(long *num*)	Devuelve el número de bits con valor en *num*.
byte byteValue( )	Devuelve el valor del objeto que invoca como un **byte**.
int compareTo(Long *l*)	Compara el valor numérico del objeto que invoca con el de *l*. Devuelve 0 si los valores son iguales. Devuelve un valor negativo si el objeto que invoca tiene menor valor. Devuelve un valor positivo si el objeto que invoca tiene mayor valor.
static Long decode(String *str*) throws NumberFormatException	Devuelve un objeto **Long** que contiene el valor especificado por la cadena *str*.
double doubleValue( )	Devuelve el valor del objeto que invoca como un **double**.
boolean equals(Object *ObjLong*)	Devuelve **verdadero** si el objeto **Long** que invoca es equivalente a *ObjLong*. De lo contrario, devuelve **falso**.
float floatValue( )	Devuelve el valor del objeto que invoca como un **float**.
static Long getLong(String *nomPropiedad*)	Devuelve el valor asociado a la propiedad de entorno especificada por *nomPropiedad*. En caso de fallo, se devuelve **null**.
static Long getLong(String *nomPropiedad*, long *om*)	Devuelve el valor asociado a la propiedad de entorno especificada por *nomPropiedad*. En caso de fallo, se devuelve el valor del parámetro *om*.
static Long getLong(String *nomPropiedad*, Long *om*)	Devuelve el valor asociado a la propiedad de entorno especificada por *nomPropiedad*. En caso de fallo, se devuelve el valor del parámetro *om*.
int hashCode( )	Devuelve el código de dispersión del objeto que invoca.
static long highestOneBit(long *num*)	Determina la posición del bits de mayor orden de *num* con valor 1. Devuelve un valor con el cual sólo este bit está definido. Si ningún bit tiene valor 1, entonces se devuelve cero.
int intValue( )	Devuelve el valor del objeto que invoca como un **int**.
long longValue( )	Devuelve el valor del objeto que invoca como un **long**.
static long lowestOneBit(long *num*)	Determina la posición del bit de menor orden definido en *num*. Devuelve un valor con el cuál solo este bit está definido. Si ningún bit tiene valor 1, entonces se devuelve cero.
static int numberOfLeadingZeros(long *num*)	Devuelve el número de bits de orden mayor en cero que preceden al primer bit de mayor orden en 1, dentro de num. Si *num* es cero, 64 es devuelto.
static int numberOfTrailingZeros(long *num*)	Devuelve el número de bits de menor orden en cero que preceden el primer bit de menor orden en 1 dentro de num. Si *num* es cero, 64 es devuelto.

TABLA 16-6 Los métodos definidos por la clase **Long**

Método	Descripción
static long parseLong(String *str*) throws NumberFormatException	Devuelve el equivalente **long** del número contenido en la cadena especificada en *str* en base 10.
static long parseLong(String *str*, int *base*) throws NumberFormatException	Devuelve el equivalente **long** del número contenido en la cadena especificada en *str* utilizando la *base* especificada.
static long reverse(long *num*)	Invierte el orden de los bits en *num* y devuelve el resultado.
static long reverseBytes(long *num*)	Invierte el orden de los bytes en *num* y devuelve el resultado.
static long rotateLeft(long *num*, int *n*)	Devuelve el resultado de rotar *num n* posiciones a la izquierda.
static long rotateRight(long *num*, int *n*)	Devuelve el resultado de rotar *num n* posiciones a la derecha.
static int signum(long *num*)	Devuelve –1 si *num* es negativo, 0 si es cero y 1 si es positivo.
short shortValue( )	Devuelve el valor del objeto que invoca como un **short**.
static String toBinaryString(long *num*)	Devuelve una cadena que contiene el equivalente binario de *num*.
static String toHexString(long *num*)	Devuelve una cadena que contiene el equivalente hexadecimal de *num*.
static String toOctalString(long *num*)	Devuelve una cadena que contiene el equivalente octal de *num*.
String toString( )	Devuelve una cadena que contiene el equivalente decimal del objeto que invoca.
static String toString(long *num*)	Devuelve una cadena que contiene el equivalente decimal de *num*.
static String toString(long *num*, int *base*)	Devuelve una cadena que contiene el decimal equivalente de *num* utilizando la *base* especificada.
static Long valueOf(long *num*)	Devuelve un objeto **Long** que contiene el valor especificado por *num*.
static Long valueOf(String *str*) throws NumberFormatException	Devuelve un objeto **Long** que contiene el valor especificado por la cadena *str*.
static Long valueOf(String *str*, int *base*) throws NumberFormatException	Devuelve un objeto **Long** que contiene el valor especificado por la cadena *str* utilizando la *base* especificada.

TABLA 16-6 Los métodos definidos por la clase **Long** (*continuación*)

Conversión entre números y cadenas

Una de las tareas más habituales en programación es convertir la representación como cadena de un número en su formato interno, binario. Afortunadamente, Java proporciona una manera fácil de hacerlo. Las clases **Byte**, **Short**, **Integer** y **Long** proporcionan los métodos **parseByte( )**,

parseShort( ), **parseInt( )** y **parseLong( )** respectivamente. Estos métodos devuelven el equivalente **byte**, **short**, **int** o **long** de la cadena numérica que los llama. Existen métodos similares para las clases **Float** y **Double**.

El siguiente programa ejemplifica el uso del método **parseInt( )**. El programa suma una lista de enteros introducidos por el usuario. Lee los enteros utilizando **readLine( )** y usa **parseInt( )** para convertir las cadenas leídas en sus valores entero equivalentes.

```
/*  Este programa suma una lista de números introducidos
    por el usuario. Convierte la representación como cadena
    de cada número en un entero utilizando el método parseInt().
*/

import java.io.*;

class ParseDemo {
  public static void main(String args[]) throws IOException {
  // crear un BufferedReader utilizando System.in
    BufferedReader br = new
      BufferedReader(new InputStrearnReader(System.in));
    String str;
    int i;
    int sum=0;

    System.out.println("Introduzca números y 0 para salir.");
    do {
      str = br.readLine();
      try {
        i = Integer.parseInt(str);
      } catch (NumberFormatException e) {
        System.out.println("Formato no válido");
        i = 0;
      }
      sum += i;
      System.out.println("La suma actual es: " + sum);
      } while(i != 0);
  }
}
```

Para convertir un número entero en una cadena decimal, han de utilizarse las versiones de **toString( )** definidas en las clases **Byte**, **Short**, **Integer** o **Long**. Las clases **Integer** y **Long** también proporcionan los métodos **toBinaryString( )**, **toHexString( )** y **toOctalString( )**, que convierten un valor en una cadena a formato binario, hexadecimal u octal, respectivamente.

El siguiente programa ejemplifica la conversión binaria, hexadecimal y octal:

```
/*  Convertir un entero en binario, hexadecimal
    y octal.
*/

class StringConversions {
  public static void main(String args[]) {
    int num = 19648;

    System.out.println(num + " en binario: " +
                       Integer.toBinaryString(num));
```

```
    System.out.println(num + " en octal: " +
                       Integer.toOctalString(num));

    System.out.println(num + " en hexadecimal: " +
                       Integer.toHexString(num));
  }
}
```

La salida del programa es ésta:

```
19648 en binario: 100110011000000
19648 en octal: 46300
19648 en hexadecimal: 4cc0
```

Character

Character es un envoltorio simple para un **char**. El constructor para **Character** es:

Character(char *ch*)

Donde *ch* especifica el carácter que será envuelto por el objeto **Character** creado.

Para obtener el valor **char** contenido en un objeto **Character**, ha de llamarse al método **charValue( )** como se muestra a continuación:

char charValue( )

El método devuelve el carácter.

La clase **Character** define diferentes constantes, incluyendo las siguientes:

MAX_RADIX	La base mayor
MIN_RADIX	La base menor
MAX_ VALUE	El valor mayor de carácter
MIN_VALUE	El valor menor de carácter
TYPE	El objeto **Class** correspondiente a **char**

La clase **Character** incluye diferentes métodos estáticos que clasifican caracteres y los convierten de mayúsculas a minúsculas o viceversa. Los métodos se muestran en la Tabla 16-7. El siguiente ejemplo muestra el uso de algunos de estos métodos.

```
// Ejemplo con varios métodos Is...

class IsDemo {
  public static void main(String args[]) {
    char a[] = {'a' , 'b' , '5' , '?', 'A', ' '};

    for(int i=0; i<a.length; i++) {
      if(Character.isDigit(a[i]))
        System.out.println(a[i] + " es un dígito.");
      if(Character.isLetter(a[i]))
        System.out.println(a[i] + " es una letra.");
      if(Character.isWhitespace(a[i]))
        System.out.println(a[i] + " es un espacio en blanco.");
```

```
      if(Character.isUpperCase(a[i]))
        System.out.println(a[i] + " es mayúscula.");
      if(Character.isLowerCase(a[i]))
        System.out.println(a[i] + " es minúscula.");
    }
  }
}
```

La salida del programa anterior es la siguiente:

```
a es una letra.
a es minúscula.
b es una letra.
b es minúscula.
5 es un dígito.
A es una letra.
A es mayúscula.
es un espacio en blanco.
```


Método	Descripción
static boolean isDefined(char *ch*)	Devuelve **verdadero** si *ch* está definida por Unicode. De lo contrario, devuelve **falso**.
static boolean isDigit(char *ch*)	Devuelve **verdadero** si *ch* es un dígito. De lo contrario, devuelve **falso**.
static boolean isIdentifierIgnorable (char *ch*)	Devuelve **verdadero** si *ch* debe ser ignorado en un identificador. De lo contrario, devuelve **falso**.
static boolean isISOControl(char *ch*)	Devuelve **verdadero** si *ch* es un carácter de control ISO. De lo contrario, devuelve **falso**.
static boolean isJavaIdentifierPart (char *ch*)	Devuelve **verdadero** si *ch* está definido como un identificador en Java (que no sea el primer carácter). De lo contrario, devuelve **falso**.
static boolean isJavaIdentifierStart(char *ch*)	Devuelve **verdadero** si *ch* es válido como primer carácter de un identificador Java. De lo contrario, devuelve **falso**.
static boolean isLetter(char *ch*)	Devuelve **verdadero** si *ch* es una letra. De lo contrario, devuelve **falso**.
static boolean isLetterOrDigit(char *ch*)	Devuelve **verdadero** si *ch* es una letra o un dígito. De lo contrario, devuelve **falso**.
static boolean isLowerCase (char *ch*)	Devuelve **verdadero** si *ch* es una letra minúscula. De lo contrario, devuelve **falso**.
static boolean isMirrored(char *ch*)	Devuelve **verdadero** si *ch* es un carácter Unicode de espejo. Un carácter de espejo es aquel que está invertido para el texto que se muestra de derecha a izquierda.

TABLA 16-7 Varios métodos de la clase **Character**

Método	Descripción
static boolean isSpaceChar (char *ch*)	Devuelve **verdadero** si *ch* es un carácter de espacio en Unicode. De lo contrario, devuelve **falso**.
static boolean isTitleCase(char *ch*)	Devuelve **verdadero** si *ch* es un carácter Unicode de título. De lo contrario, devuelve **falso**.
static boolean isUnicodeIdentifierPart(char *ch*)	Devuelve **verdadero** si *ch* puede ser parte de un identificador Unicode (que no sea el primer carácter). De lo contrario devuelve **falso**.
static boolean isUnicodeIdentifierStart(char *ch*)	Devuelve **verdadero** si *ch* puede ser el primer carácter de un identificador Unicode. De lo contrario, devuelve **falso**.
static boolean isUpperCase (char *ch*)	Devuelve **verdadero** si *ch* es una letra mayúscula. De lo contrario, devuelve **falso**.
static boolean isWhitespace (char *ch*)	Devuelve **verdadero** si *ch* es un espacio en blanco. De lo contrario, devuelve **falso**.
static char toLowerCase(char *ch*)	Devuelve el equivalente de *ch* en minúscula.
static char toTitleCase(char *ch*)	Devuelve el equivalente de *ch* en formato de título.
static char toUpperCase(char *ch*)	Devuelve el equivalente de *ch* en mayúscula.

TABLA 16-7 Varios métodos de la clase **Character** (*continuación*)

Character define los métodos **forDigit( )** y **digit( )**, que permiten convertir entre valores enteros y los dígitos que representan. Estos métodos se muestran a continuación:

static char forDigit(int *num*, int *base*)
static int digit(char *digit*, int *base*)

forDigit( ) devuelve el carácter (un dígito) asociado al valor de *num*. La base de la conversión se especifica en *base*. **digit( )** devuelve el valor entero asociado al carácter especificado (que presumiblemente es un dígito) de acuerdo con la base especificada.

Otro método definido por **Character** es **compareTo( )**, que tiene la siguiente forma:

int compareTo(Character *c*)

Devuelve cero si el objeto que invoca y *c* tienen el mismo valor. Devuelve un valor negativo si el objeto que invoca tiene un valor menor, de lo contrario, devuelve un valor positivo.

Character incluye un método llamado **getDirectionality( )** el cual puede ser utilizado para determinar la dirección del carácter. Muchas constantes están definidas para describir direccionamiento. La mayoría de los programas no necesitarán hacer uso del direccionamiento de caracteres.

La clase **Character** también sobrescribe los métodos **equals( )** y **hashCode( ).**

Dos clases relacionadas con caracteres son **Character.Subset**, utilizada para describir un subconjunto de Unicode, y **Character.UnicodeBlock**, que contiene bloques de caracteres Unicode.

Adiciones recientes al tipo character para soporte de Unicode

Recientemente, importantes adiciones se han hecho a la clase **Character**. A partir del JDK 5, se ha incluido en la clase **Character** soporte para caracteres Unicode de 32 bits. En el pasado, todos los caracteres Unicode podían almacenarse con 16 bits, lo cual es el tamaño de un **char** (y el tamaño del valor encapsulado dentro de un **Character**), dado que éste es el rango de valores desde 0 hasta FFFF. Sin embargo, los caracteres Unicode han sido extendidos, y se requieren más de 16 bits. La clase Characters puede ahora tener un rango que va desde 0 hasta 10FFFF.

A continuación se mencionan dos términos importantes: apuntador a código y carácter suplementario. Un *apuntador a código* es un carácter en el rango de 0 a 10FFFF. Los caracteres que tienen valores mayores que FFFF son denominados *caracteres suplementarios*.

La expansión del conjunto de caracteres Unicode causó un problema fundamental para Java. Dado que los caracteres suplementarios tienen un valor mayor del que una variable **char** puede contener, se hizo necesario definir una forma para dar soporte a los caracteres suplementarios. Java dirigió este problema de dos formas. En primer lugar, Java utilizó dos **char** para representar un carácter suplementario. El primer **char** fue llamado *alto sustituto* y el segundo fue llamado *bajo sustituto*. Los métodos nuevos, tales como **codePointAt( )**, fueron suministrados para traducir entre apuntadores a código y caracteres suplementarios.

En segundo lugar, Java sobrescribió muchos de los métodos preexistentes en la clase **Character**. Los métodos sobrescritos utilizan un dato de tipo **int** en lugar de uno de tipo **char**. Dado que un **int** es lo suficientemente largo como para contener cualquier carácter en un solo valor, éste puede ser utilizado para almacenar cualquier carácter. Por ejemplo, todos los métodos de la Tabla 16-7 tienen la forma sobrescrita que opera con **int**. A continuación un ejemplo:

```
static boolean isDigit(int cp)
static boolean isLetter(int cp)
static int toLowerCase(int cp)
```

Además de los métodos sobrescritos para aceptar apuntadores de código, **Character** agrega métodos que proveen soporte adicional para apuntadores de código. Algunos ejemplos se muestran en la Tabla 16-8.

Método	Descripción
static int charCount(int *cp*)	Devuelve 1 si *cp* puede ser representado por un **char**. Devuelve 2 si se necesitan dos **char**.
static int codePointAt (CharSequence *chars*, int *loc*)	Devuelve el apuntador a código, dirigido a la localidad especificada en *loc*.
static int codePointAt(char *chars*[], int *loc*)	Devuelve el apuntador a código, dirigido a la localidad especificada en *loc*.
static int codePointBefore(CharSequence *chars*, int *loc*)	Devuelve el apuntador a código dirigido a la localidad previa a la especificada en *loc*.
static int codePointBefore(char *chars*[], int *loc*)	Devuelve el apuntador a código dirigido a la localidad previa a la especificada en *loc*.
static boolean isHighSurrogate(char *ch*)	Devuelve **verdadero** si *ch* contiene un carácter alto substituto válido.

TABLA 16-8 Ejemplo de métodos que proveen soporte para apuntadores de código Unicode de 32 bits

Método	Descripción
static boolean isLowSurrogate(char *ch*)	Devuelve **verdadero** si *ch* contiene un carácter bajo sustituto válido.
static boolean isSupplementaryCodePoint(int *cp*)	Devuelve **verdadero** si *cp* contiene un carácter suplementario.
static boolean isSurrogatePair(char *highCh*, char *lowCh*)	Devuelve **verdadero** si *highCh* y *lowCh* forman un par sustituto válido.
static boolean isValidCodePoint(int *cp*)	Devuelve **verdadero** si *cp* contiene un apuntador a código válido.
static char[ ] toChars (int *cp*)	Convierte el apuntador de código en *cp* a su **char** equivalente, el cual podría requerir dos **char**. Devuelve un arreglo con los resultados.
static int toChars(int *cp*, char *target*[ ],int *loc*)	Convierte el apuntador de código en *cp* a su **char** equivalente, almacenando el resultado en *target*, comenzando en *loc*. Devuelve 1 si *cp* puede ser representado por un solo **char**. Devuelve 2 en cualquier otro caso.
static int toCodePoint(char *highCh*, char *lowCh*)	Convierte *highCh* y *lowCh* en sus equivalentes apuntadores a código.

TABLA 16-8 Ejemplo de métodos que proveen soporte para Unicode de 32 bits (*continuación*)

Boolean

Boolean es un envoltorio muy fino para valores **boolean**, que es útil sobre todo cuando se quiere pasar una variable **boleana** por referencia. Contiene las constantes **TRUE** y **FALSE**, que definen los objetos **boleanos** verdadero y falso respectivamente. **Boolean** también define el campo **TYPE**, que es el objeto **Class** para **boolean**. **Boolean** define estos constructores:

Boolean(boolean *valorBool*)
Boolean(String *cadenaBool*)

En la primera versión, *valorBool* debe ser **true** o **false**. En la segunda, si *cadenaBool* contiene la cadena "true" (en mayúsculas o minúsculas), entonces el nuevo objeto **Boolean** será true; de lo contrario, será false.

Boolean define los métodos mostrados en la Tabla 16-9.

Método	Descripción
boolean booleanValue( )	Devuelve el valor equivalente de tipo **boolean**.
int compareTo(Boolean *b*)	Devuelve cero si el objeto que invoca y *b* contienen el mismo valor. Devuelve un valor positivo si el objeto que invoca es **verdadero** y *b* es **falso**. En otro caso, devuelve un valor negativo.
boolean equals(Object *objBool*)	Devuelve **verdadero** si el objeto que invoca es equivalente a *objBool*. De lo contrario, devuelve **falso**.

TABLA 16-9 Los métodos definidos por la clase **Boolean**

Método	Descripción
static boolean getBoolean(String *nomProp*)	Devuelve **verdadero** si la propiedad del sistema especificada por *nomProp* tiene el valor **verdadero**. De lo contrario, devuelve **falso**.
int hashCode( )	Devuelve el código de dispersión del objeto que invoca.
static boolean parseBoolean(String *str*)	Devuelve **verdadero** si *str* contiene la cadena "true". Sin importar mayúsculas o minúsculas. De lo contrario devuelve **falso**.
String toString( )	Devuelve la cadena equivalente del objeto que invoca.
static String toString(boolean *boolVal*)	Devuelve la cadena equivalente a *boolVal*.
static Boolean valueOf(boolean *boolVal*)	Devuelve el objeto **Boolean** equivalente a *boolVal*.
static Boolean valueOf(String *cadenaBool*)	Devuelve **verdadero** si *cadenaBool* contiene la cadena "true" no importa si es mayúsculas o minúsculas. De lo contrario, devuelve **falso**.

TABLA 16-9 Los métodos definidos por la clase **Boolean** (*continuación*)

Void

La clase **Void** tiene un campo, **TYPE**, que contiene una referencia al objeto de tipo **Class** para el tipo **void**. No se crean instancias de esta clase.

La clase Process

La clase abstracta **Process** encapsula un *proceso*, esto es, un programa en ejecución. Se utiliza básicamente como una superclase para el tipo de objetos creados por el método **exec( )** de la clase **Runtime**, o por el método **start( )** en la clase **ProcessBuilder**. La clase **Process** contiene los métodos abstractos mostrados en la Tabla 16-10.

Método	Descripción
void destroy( )	Termina el proceso.
int exitValue( )	Devuelve un código de salida obtenido de un subproceso.
InputStream getErrorStream( )	Devuelve un flujo de entrada que lee la entrada desde el flujo de salida **err** del proceso.
InputStream getInputStream( )	Devuelve un flujo de entrada que lee la entrada desde el flujo de salida **out** del proceso.
OutputStream getOutputStream( )	Devuelve un flujo de salida que escribe la salida al flujo de salida **in** del proceso.
int waitFor( ) throws InterruptedException	Devuelve el código de salida devuelto por el proceso. Este método no regresa hasta que no termina el proceso desde el que se le llama.

TABLA 16-10 Los métodos definidos por la clase **Process**

La clase Runtime

La clase **Runtime** encapsula al proceso del intérprete de Java que se ejecuta. No se puede crear una instancia de la clase **Runtime**. Sin embargo, se puede obtener una referencia al objeto **Runtime** actual mediante una llamada al método estático **Runtime.getRuntime( )**. Una vez obtenida la referencia al objeto **Runtime** actual, se puede llamar a diversos métodos que controlan el estado y comportamiento de la Máquina Virtual de Java. Normalmente, los applets y otros fragmentos de código no fiables no pueden llamar a ninguno de los métodos **Runtime** sin producir una **SecurityException** (excepción de seguridad). Entre los métodos definidos por la clase **Runtime** comúnmente utilizados se encuentran los mostrados en la Tabla 16-11.

Método	Descripción
void addShutdownHook(Thread *hilo*)	Registra *hilo* como un hilo a correr cuando la máquina virtual Java (JVM) termina.
Process exec(String *nomProg*) throws IOException	Ejecuta el programa especificado por *nomProg* como un proceso separado. Se devuelve un objeto de tipo **Process** que describe al nuevo proceso.
Process exec(String *nomProg*, String *entorno*[ ]) throws IOException	Ejecuta el programa especificado por *nomProg* como un proceso separado con el *entorno* especificado en la variable entorno. Se devuelve un objeto de tipo **Process** que describe al nuevo proceso.
Process exec (String *arregloCom*[ ]) throws IOException	Ejecuta la línea de comandos especificada por las cadenas contenidas en *arregloCom* como un proceso separado. Se devuelve un objeto de tipo **Process** que describe al nuevo proceso.
Process exec (String *arregloCom*[ ], String *entorno*[ ]) throws IOException	Ejecuta la línea de comandos especificada por las cadenas de *arregloCom* como un proceso separado con el entorno especificado en la variable *entorno*. Se devuelve un objeto de tipo **Process** que describe al nuevo proceso.
void exit(int *codigoSalida*)	Detiene la ejecución y devuelve el valor de *codigoSalida* al proceso padre. Por convención, 0 indica terminación normal. Todos los demás valores indican alguna forma de error.
long freeMemory( )	Devuelve el número aproximado de bytes de memoria libre disponible para el sistema de ejecución de Java.
void gc( )	Inicia la recolección de basura.
static Runtime getRuntime( )	Devuelve el objeto **Runtime** actual.
void halt(int *code*)	Termina inmediatamente la ejecución de la Máquina Virtual de Java. No se corren hilos de terminación ni finalizadores. El valor del parámetro *c* se devuelve al proceso que invoca.
void load(String *archivo*)	Carga la biblioteca dinámica cuyo archivo se indica en el parámetro *archivo*, que debe especificar el nombre y la ruta completa del archivo.
void loadLibrary(String *nombre*)	Carga la biblioteca dinámica cuyo nombre está asociado al parámetro *nombre*.

TABLA 16-11 Los métodos de uso común definidos por la clase **Runtime**

Método	Descripción
boolean removeShutdownHook (Thread *hilo*)	Elimina *hilo* de la lista de hilos a ejecutar cuando la Máquina Virtual de Java termina. Devuelve el valor **true** en caso de éxito, esto es, si el hilo fue eliminado.
void runFinalization( )	Inicia llamadas a los métodos **finalize( )** de objetos no utilizados pero todavía no reciclados.
long totalMemory( )	Devuelve el número total de bytes de memoria disponible para el programa.
void traceInstructions (boolean *rastreo*)	Activa o desactiva el rastreo de instrucciones, dependiendo del valor del parámetro *rastreo*. Si *rastreo* tiene el valor **true**, el rastreo se muestra. Si tiene el valor **false**, el rastreo se desactiva.
void traceMethodCalls(boolean *rastreo*)	Activa o desactiva el rastreo de llamadas a métodos, dependiendo del valor del parámetro *rastreo*. Si *rastreo* tiene el valor **true**, el rastreo se muestra. Si tiene el valor **false**, el rastreo se desactiva.

TABLA 16-11 Los métodos de uso común definidos por la clase **Runtime** (*continuación*)

Veamos dos de los usos más comunes de la clase **Runtime**: administración de memoria y ejecución de procesos adicionales.

Administración de memoria

Aunque Java proporciona una recolección automática de basura (más conocida por su nombre en inglés garbage collection) en ocasiones nos interesara conocer el tamaño actual de la pila de objetos y el espacio que queda libre. Se puede utilizar esta información, por ejemplo, para comprobar la eficiencia del código o para saber cuántos objetos más de cierto tipo pueden ser instanciados. Para obtener estos valores se utilizan los métodos **totalMemory( )** y **freeMemory( )**.

Tal como hemos mencionado en la Parte I, la recolección de basura de Java se ejecuta periódicamente para reciclar objetos no utilizados. Sin embargo, en ocasiones nos interesa recoger objetos que no están siendo utilizados antes de la siguiente ejecución de la recolección automática. Se puede ejecutar la recolección automática de basura a voluntad llamando al método **gc( )**. Es buena práctica llamar al método **gc( )** y en seguida al método **freeMemory( )** para obtener una referencia base del uso de memoria. Luego ejecutamos nuestro código y llamamos de nuevo al método **freeMemory( )** para ver, por comparación, cuánta memoria estamos asignando. El siguiente programa ilustra esta idea:

```
// Ejemplo con totalMemory(), freeMemory() y gc( )

class MemoriaDemo{
  public static void main(String args[]) {
    Runtime r = Runtime.getRuntime();
    long meml, mem2;
    Integer algunosenteros[] = new Integer[l000];

    System.out.println("Memoria total: " +
                       r.totalMemory());
    meml = r.freeMemory();
```

```
    System.out.println("Memoria libre inicial: " + meml);
    r.gc( );
    meml = r.freeMemory();
    System.out.println("Memoria libre tras la recolección de basura: "
                       + meml);

    for(int i=0; i<l000; i++)
      algunosenteros[i] = new Integer(i); // asignar enteros

    mem2 = r. freeMemory() ;
    System.out.println("Memoria libre tras la asignación: "
                       + mem2);

    System.out.println("Memoria utilizada por la asignación: "
                       + (meml-mem2));

    // descartar enteros
    for(int i=0; i<1000; i++) algunosenteros[i] = null;

    r.gc(); //solicitar recolección de basura

    mem2 = r.freeMemory() ;
    System.out.println("Memoria libre tras recoger" +
                       " enteros descartados: " + mem2);

  }
}
```

A continuación se muestra una posible salida de este programa (por supuesto, los resultados pueden variar en cada caso):

```
Memoria total: 1048568
Memoria libre inicial: 751392
Memoria libre tras la recolección de basura: 841424
Memoria libre tras la asignación: 824000
Memoria utilizada por la asignación: 17424
Memoria libre tras recoger enteros descartados: 842640
```

Ejecución de otros programas

En entornos seguros, se puede utilizar Java para ejecutar otros procesos pesados (es decir, programas) bajo un sistema operativo multitarea. Diferentes formas del método **exec( )** permiten especificar el nombre del programa que se desea ejecutar junto con sus parámetros de entrada. El método **exec( )** devuelve un objeto **Process**, que se puede utilizar para controlar cómo interactúa el programa en Java con este nuevo proceso en ejecución. Como Java puede correr en diferentes plataformas y bajo diferentes sistemas operativos, **exec( )** es intrínsecamente dependiente del entorno.

El siguiente ejemplo utiliza al método **exec( )** para abrir **notepad**, el sencillo editor de texto de Windows. Obviamente, este ejemplo debe ejecutarse bajo el sistema operativo Windows.

```
// Ejemplo con el método exec().
  class ExecDemo{
    public static void main(String args[]) {
      Runtime r = Runtime.getRuntime();
      Process p = null;
    try {
      p = r.exec("notepad");
```

```
    } catch (Exception e) {
      System.out.println("Error al ejecutar notepad.");
    }
  }
}
```

Existen diferentes formas alternativas del método **exec( )**, pero la mostrada en el ejemplo es la más común. El objeto Process devuelto por el método **exec( )** puede manipularse mediante los métodos de la clase Process después de que el nuevo programa inicia su ejecución. Se puede eliminar el subproceso con el método **destroy( )**. El método **waitFor( )** hace que el programa Java espere hasta la terminación del subproceso. El método **exitValue( )** devuelve el valor devuelto por el subproceso cuando éste termina, que es típicamente 0 si no hay problemas. A continuación se muestra el ejemplo anterior del método **exec( )** modificado para esperar a que el proceso en ejecución termine:


```
// Esperar hasta que notepad termine.
class ExecDemoFini {
  public static void main(String args[]) {
    Runtime r = Runtime.getRuntime();
    Process p = null;

    try {
      p = r.exec("notepad");
      p.waitFor();
    } catch (Exception e) {
      System.out.println("Error ejecutando notepad.");
    }
    System.out.println("Notepad ha devuelto" + p.exitValue());
  }
}
```

Mientras un subproceso corre, se puede escribir y leer en su entrada y salida estándar. Los métodos **getOutputStream( )** y **getInputStream( )** devuelven los descriptores de la **entrada** y la **salida** estándar del subproceso (el tema de E/S se trata en detalle en el Capítulo 19).

La clase ProcessBuilder

ProcessBuilder provee otra forma de comenzar y administrar procesos (es decir, programas). Como se explicó anteriormente, todos los procesos son representados por la clase **Process**, y un proceso puede ser iniciado por **Runtime.exec( )**. La clase **ProcessBuilder** ofrece mayor control sobre los procesos. Por ejemplo, se puede definir el directorio actual de trabajo y cambiar los parámetros de ambiente.

La clase **ProcessBuilder** define estos constructores:

ProcessBuilder(List <String> args)
ProcessBuilder(String … args)

Donde, *args* es una lista de argumentos que especifican el nombre del programa a ser ejecutado junto con cualquier argumento de línea de comandos requerida. En el primer constructor, los argumentos son pasados en un objeto de tipo **List**. En el segundo, se especifican a través de parámetros varargs. La Tabla 16-12 describe los métodos definidos por la clase **ProcessBuilder**.

Para crear un proceso utilizando la clase **ProcessBuilder**, simplemente se crea una instancia de **ProcessBuilder**, especificando el nombre del programa y cualquier argumento que se necesite. Para comenzar la ejecución del programa, se llama al método **start( )** sobre la instancia. A continuación un ejemplo que ejecuta el editor de textos **notepad** de Windows. Note que se especifica el nombre del archivo a editar como un argumento.

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

    try {
      ProcessBuilder proc =
        new ProcessBuilder("notepad.exe", "archivoPrueba");
      proc.start();
    } catch (Exception e){
      System.out.println("Error ejecutando notepad");
    }
  }
}
```

Método	Descripción
List<String> command( )	Devuelve una referencia a un objeto **List** que contiene el nombre del programa y sus argumentos. Los cambios en esta lista afectan al proceso que invoca.
ProcessBuilder command(List<String> *args*)	Define el nombre del programa y sus argumentos con lo especificado por *args*. Los cambios a esta lista afecta al proceso que invoca. Devuelve una referencia al objeto que invoca.
ProcessBuilder command(String ... *args*)	Define el nombre del programa y sus argumentos con lo especificado por *args*. Devuelve una referencia al objeto que invoca.
File directory( )	Devuelve el directorio de trabajo actual del objeto que invoca. Este valor será **null** si el directorio es el mismo que el del programa de Java que comenzó al proceso.
ProcessBuilder directoy(File *dir*)	Define el directorio actual de trabajo del objeto que invoca. Devuelve una referencia al objeto que invoca.
Map<String, String> environment( )	Devuelve las variables de ambiente asociadas con el objeto que invoca como pares clave/valor.
boolean redirectErrorStream( )	Devuelve **verdadero** si el flujo de error estándar ha sido redireccionado al flujo de salida estándar. Devuelve **falso** si los flujos están separados.
ProcessBuilder redirectErrorStream(boolean *fusion*)	Si *fusion* es **verdadero**, entonces el flujo de error estándar es redireccionado a la salida estándar. Si *fusion* es **falso**, los flujos son separados, éste es el estado por omisión. Devuelve una referencia al objeto que invoca.
Process start( ) throws IOException	Comienza al proceso especificado por el objeto que invoca. En otras palabras, ejecuta el programa especificado.

TABLA 16-12 Los métodos definidos por la clase **ProcessBuilder**

La clase System

La clase **System** contiene una colección de métodos y variables estáticos. La entrada, salida y salida de errores estándar del intérprete de Java se almacenan en las variables **in**, **out** y **err** respectivamente. Los métodos definidos por **System** se muestran en la Tabla 16-13. Nótese que muchos de los métodos arrojan una excepción del tipo **SecurityException** si la operación no está permitida por el gestor de seguridad.

Veamos algunos usos comunes de **System**.

Método	Descripción
static void arraycopy(Object *fuente*, int *inicioFuente*, Object *destino*, int *inicioDestino*, int *tamaño*)	Copia un arreglo. El arreglo a ser copiado se pasa en *fuente*, y el índice del punto en que comenzará la copia dentro de *fuente* se pasa en *inicioFuente*. El arreglo que recibirá la copia se pasa en *destino*, y el índice del punto donde comenzará la copia dentro de *destino* se pasa en *inicioDestino*. El número de elementos que se copiarán se especifica en tamaño.
static String clearProperty(String *v*)	Elimina la variable de ambiente especificada por *v*. El valor previo asociado con *v* es devuelto.
static Console console( )	Devuelve la consola asociada con la JVM. Se devuelve **null** si la JVM actual no tiene consola (agregado por Java SE 6).
static long currentTimeMillis( )	Devuelve la hora actual en milisegundos desde la medianoche del 1 de enero de 1970.
static void exit(int *codigoSalida*)	Detiene la ejecución y devuelve el valor de *codigoSalida* al proceso padre (habitualmente el sistema operativo). Por convención, 0 indica terminación normal. Todos los otros valores indican algún tipo de error.
static void gc( )	Inicia la recolección de basura.
static Map<String, String> getenv( )	Devuelve un objeto **Map** que contiene las variables de ambiente actuales y sus valores.
static String getenv(String *v*)	Devuelve el valor asociado con la variable de ambiente especificada por *v*.
static Properties getProperties( )	Devuelve las propiedades asociadas con el intérprete de Java. (La clase **Properties** se describe en el Capítulo 17).
static String getProperty(String *p*)	Devuelve la propiedad asociada a *p*. Un objeto **null** se devuelve si la propiedad deseada no es encontrada.
static String getProperty(String *p*, String *om*)	Devuelve la propiedad asociada con *p*. Si la propiedad deseada no se encuentra, se devuelve el valor especificado en *om*.
static SecurityManager getSecurityManager( )	Devuelve el gestor de seguridad en uso o un objeto **null** si no hay un gestor de seguridad instalado.
static int identityHashCode (Object *obj*)	Devuelve la identidad del código de dispersión para *obj*.

TABLA 16-13 Los métodos definidos por la clase **System**

Método	Descripción
static Channel inheritedChannel( ) thows IOException	Devuelve el canal heredado por la Máquina Virtual de Java. Devuelve **null** si no se hereda ningún canal.
static void load(String *nombreArchivo*)	Carga la biblioteca dinámica contenida en el archivo especificado por *nombreArchivo*; *nombreArchivo* debe especificar la ruta completa del archivo.
static void loadLibrary(String *nomBiblioteca*)	Carga la biblioteca dinámica cuyo nombre está asociado con *nomBiblioteca*.
static String mapLibraryName (String *b*)	Devuelve el nombre en una plataforma específica para la biblioteca llamada *b*.
static long nanoTime( )	Obtiene un tiempo cronometrado de la manera más precisa posible en el sistema y devuelve su valor en términos de nanosegundos comenzando desde algún punto arbitrario. La exactitud del tiempo medido es impredecible.
static void runFinalization( )	Inicia llamadas a los métodos **finalize()** de objetos no utilizados y que no han sido reciclados.
static void setErr(PrintStream *flujoErr*)	Establece como el flujo estándar de **error** a *flujoErr*.
static void setIn(PrintStream *flujoEnt*)	Establece como el flujo estándar de **entrada** a *flujoEnt*.
static void setOut(PrintStream *flujoSal*)	Establece como el flujo estándar de **salida** a *flujoSal*.
static void setProperties(Properties *p*)	Establece las propiedades actuales del sistema a los valores especificados por *p*.
static String setProperty(String *p*, String *v*)	Asigna el valor *v* a la propiedad llamada *p*.
static void setSecurity Manager (SecurityManager *s*)	Establece el gestor de seguridad al indicado por el objeto *s*.

TABLA 16-13 Los métodos definidos por la clase **System** (*continuación*)

Uso de currentTimeMillis( )

Un uso de la clase **System** que puede ser de particular interés es el uso del método **currentTimeMillis( )** para medir cuánto tardan en ejecutarse diversas partes del programa. El método **currentTimeMillis( )** devuelve la hora actual en milisegundos desde la medianoche del 1 de enero de 1970. Para cronometrar una parte del programa se almacena este valor justo antes del comienzo de la parte en cuestión; inmediatamente después de terminar su ejecución, se llama a **currentTimeMillis( )** de nuevo. El tiempo transcurrido será entonces el valor obtenido al terminar, menos el almacenado al comenzar. El siguiente programa lo ejemplifica:

```
// Cronometrando la ejecución de un programa.

class Elapsed {
  public static void main(String args[]) {
    long inicio, fin;

    System.out.println("Cronometrando un ciclo de 0 a 1,000,000");

    // tiempo transcurrido en un ciclo de 0 a 1,000,000
```

```
    inicio = System.currentTimeMillis(); // tiempo inicial
    for(int i=0; i < 1000000; i++) ;
    fin = System.currentTimeMillis(); // tiempo final

    System.out.println("Tiempo en milisegundos: " + (fin-inicio));
  }
}
```

La siguiente es una posible salida de la ejecución del programa (recuérdese que los resultados podrán variar cada vez):

```
Cronometrando un ciclo de 0 a 1,000,000
Tiempo en milisegundos: 10
```

Si el sistema tiene un cronómetro que ofrece precisión en nanosegundos, entonces se podría sobrescribir el código anterior para usar **nanoTime( )** en lugar de **currentTimeMillis( )**. Por ejemplo, a continuación está la porción clave del programa anterior, reescrita para utilizar **nanoTime( )**:

```
start = System.nanoTime(); // tiempo inicial
for (int i=0; i < 1000000; i++);
fin = System.natoTime(); // tiempo final
```

Uso de arraycopy( )

El método **arraycopy( )** se puede utilizar para copiar rápidamente un arreglo de cualquier tipo de un sitio a otro. Esto es mucho más rápido que utilizar un ciclo equivalente, escrito a mano en Java. Aquí tenemos un ejemplo de dos arreglos que se copian mediante el método **arraycopy( )**. Primero, se copia el arreglo **a** en el arreglo **b**. Después, todos los elementos de **a** se desplazan una posición hacia abajo, y por último, los elementos de **b** una posición hacia arriba.

```
// Ejemplo con arraycopy().

class ACDemo {
  static byte a[] = { 65, 66, 67, 68, 69, 70, 71, 72, 73, 74 };
  static byte b[] = { 77, 77, 77, 77, 77, 77, 77, 77, 77, 77 };

  public static void main(String args[]) {
    System.out.println("a = " + new String(a));
    System.out.println("b = " + new String(b));
    System.arraycopy(a, 0, b, 0, a.length);
    System.out.println("a = " + new String(a));
    System.out.println("b = " + new String (b));
    System.arraycopy(a, 0, a, 1, a.length - 1);
    System.arraycopy(b, 1, b, 0, b.length - 1);
    System.out.println("a = " + new String(a));
    System.out.println("b = " + new String(b));
  }
}
```

Como se ve en la siguiente salida, se puede copiar en cualquier dirección utilizando la misma fuente y el mismo destino:

```
a = ABCDEFGHIJ
b = MMMMMMMMM
a = ABCDEFGHIJ
```

```
b = ABCDEFGHIJ
a = AABCDEFGHI
b = BCDEFGHIJJ
```

Propiedades del entorno

Las siguientes propiedades están disponibles:

file.separator	java.specification.version	line.separator
java.class.path	java.vendor	os.arch
java.class.version	java.vendor.url	os.name
java.compiler	java.version	os.version
java.ext.dirs	java.vm.name	path.separator
java.home	java.vm.specification.name	user.dir
java.io.tmpdir	java.vm.specification.vendor	user.home
java.library.path	java.vm.specification.version	user.name
java.specification.name	java.vm.vendor	
java.specification.vendor	java.vm.version	

Es posible obtener los valores de las diversas variables de entorno llamando al método **System.getProperty( )**. Por ejemplo, el siguiente programa muestra el directorio actual del usuario:

```
class ShowUserDir {
  public static void main(String args[]) {
    System.out.println(System.getProperty("user.dir"));
  }
}
```

La clase Object

Como mencionamos en la Parte I, **Object** es una superclase de todas las demás clases. **Object** define los métodos mostrados en la Tabla 16-14, los cuales están disponibles para todos los objetos.

Método	Descripción
Object clone( ) throws CloneNotSupportedException	Crea un nuevo objeto que es igual que el objeto que invoca.
boolean equals(Object *objeto*)	Devuelve **verdadero** si el objeto que invoca es equivalente a *objeto*.
void finalize( ) throws Throwable	Método **finalize( )** por omisión. Normalmente es sobrescrito por las subclases.

TABLA 16-14 Los métodos definidos por la clase **Object**

Método	Descripción
final Class<?>getClass( )	Obtiene un objeto **Class** que describe al objeto que invoca.
int hashCode( )	Devuelve el código de dispersión asociado al objeto que invoca.
final void notify( )	Reanuda la ejecución de un hilo en espera del objeto que invoca.
final void notifyAll( )	Reanuda la ejecución de todos los hilos en espera del objeto que invoca.
String toString( )	Devuelve una cadena que describe el objeto.
final void wait( ) throws InterruptedException	Espera a otro hilo de ejecución
final void wait(long *milisegundos*) throws InterruptedException	Espera a otro hilo de ejecución durante el número de *milisegundos* indicado.
final void wait(long *milisegundos*, int *nanosegundos*)throws InterruptedException	Espera a otro hilo de ejecución durante el número de *milisegundos* más *nanosegundos* indicados.

TABLA 16-14 Los métodos definidos por la clase **Object** (*continuación*)

El método clone( ) y la interfaz Cloneable

La mayoría de los métodos definidos por **Object** se tratan a lo largo de este libro. Sin embargo, uno de ellos merece especial atención: el método **clone( )**. El método **clone( )** genera un duplicado del objeto sobre el que se llama. Sólo se pueden clonar clases que implementen la interfaz **Cloneable**.

La interfaz **Cloneable** no define ningún miembro. Se usa para indicar que una clase permite la realización de una copia bit a bit de un objeto (esto es, un *clon*). Si se intenta llamar al método **clone( )** sobre una clase que no implementa Cloneable, se produce una excepción de tipo **CloneNotSupportedException**. Cuando se hace un clon, *no* se invoca al método constructor del objeto en clonación. Un clon es simplemente una copia exacta del original.

La clonación es una acción potencialmente peligrosa, porque puede ocasionar efectos colaterales no deseados. Por ejemplo, si el objeto clonado contiene una referencia en una variable llamada *refOb*, entonces cuando se hace el clon, *refOb* en el clon hace referencia al mismo objeto que *refOb* en el original. Si el clon hace un cambio en los contenidos del objeto referenciado por *refOb*, entonces quedará cambiado también para el objeto original. Otro ejemplo: si un objeto abre un flujo de E/S y luego se clona, habrá dos objetos capaces de operar sobre el mismo flujo. Además, si uno de esos objetos cierra el flujo, el otro podría intentar escribir en él, causando un error. En algunos casos se necesitará sobrescribir el método **clone( )** definido por la clase **Object** para gestionar este tipo de problemas.

Puesto que la clonación puede causar problemas, **clone( )** se declara como **protected** dentro de **Object**. Esto significa que debe ser llamado desde un método definido por una clase que implemente la interfaz **Cloneable**, o bien debe ser sobrescrito explícitamente por esa clase para que sea público. Veamos un ejemplo de cada uno de estos dos casos.

El siguiente programa implementa **Cloneable** y define el método **cloneTest( )**, que llama al método **clone( )** de **Object**.

```
// Ejemplo con el método clone()

class TestClone implements Cloneable {
  int a;
  double b;

  // Este método llama a clone() de Object.
  TestClone cloneTest() {
    try {
    // llama a clone en Object.
    return (TestClone) super.clone();
    } catch(CloneNotSupportedException e) {
      System.out.println("Clonación no permitida.");
      return this;
    }
  }
}

class CloneDemo {
  public static void main(String args[]) {
    TestClone xl = new TestClone();
    TestClone x2;

    x1.a =10;
    x1.b = 20.98;

    x2 =xl.cloneTest(); // clonación de xl

    System.out.println("xl: " + xl.a + " " + x1.b);
    System.out.println("x2: " + x2.a + " " + x2.b);
  }
}
```

Aquí, el método **cloneTest( )** llama al método **clone( )** de la clase **Object** y devuelve el resultado. Nótese que el objeto devuelto por **clone( )** debe convertirse en su tipo apropiado (**TestClone**).

El siguiente ejemplo sobrescribe **clone( )** para que pueda ser llamado desde código fuera de su clase. Para hacer esto, su especificador de acceso debe ser **public**, como en este ejemplo:

```
// Sobrescribe el método clone().

class TestClone implements Cloneable {
  int a;
  double b;

  // clone() está ahora sobrescrito y es público.
  public Object clone() {
    try(
      // llama a clone en Object.
      return super.clone();
    } catch(CloneNotSupportedException e) {

      System.out.println("Clonación no permitida.");
      return this;
    }
  }
}
```

```
class CloneDemo2 {
  public static void main(String args[]) {
    TestClone x1 = new TestClone();
    TestClone x2;

    x1.a = 10;
    x1.b = 20.98;

    // aquí se llama a clone() directamente.
    x2 = (TestClone) x1.clone();

    System.out.println("x1: " + x1.a + " " + x1.b);
    System.out.println("x2: " + x2.a + " " + x2.b);
  }
}
```

Los efectos colaterales causados por la clonación son difíciles de detectar a primera vista. Es fácil pensar que una clase es segura para la clonación cuando de hecho no lo es. En general, es mejor no implementar **Cloneable** para ninguna clase si no hay una buena razón para ello.

Class

Class encapsula el estado en tiempo de ejecución de un objeto o interfaz. Los objetos del tipo **Class** se crean automáticamente, cuando se cargan las clases. No se puede declarar explícitamente un objeto **Class**. Generalmente, se obtiene un objeto **Class** llamando al método **getClass( )** definido por **Object**.

Class es un tipo genérico que se declara como se muestra a continuación:

class Class <T>

Donde, **T** es el tipo de la clase o interfaz representada. Un ejemplo de los métodos definidos por **Class** se muestran en la Tabla 16-15

Método	Descripción
static Class <?> forName(String *nombre*) throws ClassNotFoundException	Devuelve un objeto **Class** dando su nombre completo.
static Class<?> forName(String *nombre*, boolean *c*, ClassLoader *cgr*) throws ClassNotFoundException	Devuelve un objeto **Class**, dando su nombre completo. El objeto se carga utilizando el cargador especificado en *cgr*. Si el parámetro *c* es **verdadero**, el objeto es inicializado.
<A extends Annotation> A getAnnotation(Class <A> *aTipo*)	Devuelve un objeto **Annotation** que contiene la anotación asociada con *aTipo* para el objeto invocado.

TABLA 16-15 Algunos métodos definidos por la clase **Class**

Método	Descripción
Annotation[ ] getAnnotations( )	Obtiene todas las anotaciones asociadas con el objeto que invoca y las almacena en un arreglo de objetos de tipo **Annotation**. Devuelve una referencia a este arreglo.
Class<?>[ ] getClasses( )	Devuelve un objeto **Class** para cada una de las clases e interfaces públicas que son miembros del objeto que invoca.
ClassLoader getClassLoader( )	Devuelve el objeto **ClassLoader** que cargó la clase o interfaz utilizada para instanciar al objeto que invoca.
Constructor<T> getConstructors (Class<?> ... *pTipos*) throws NoSuchMethodException, SecurityException	Devuelve un objeto de tipo **Constructor** que representa al constructor del objeto que invoca que tiene los tipos de parámetros especificados por *pTipos*.
Constructor<?>[ ] getConstructors( ) throws SecurityException	Obtiene un objeto de tipo **Constructor** para cada constructor público del objeto que invoca y los almacena en un arreglo. Devuelve la referencia a dicho arreglo.
Annotation[ ] getDeclaredAnnotations( )	Obtiene un objeto de tipo **Annotation** para todas las anotaciones que están declaradas por el objeto que invoca y los almacena en un arreglo. Devuelve una referencia a dicho arreglo (las anotaciones heredadas son ignoradas).
Constructor<?>[ ] getDeclared Constructors( )throws SecurityException	Obtiene un objeto de tipo **Constructor** para cada constructor declarado por el objeto que invoca y los almacena en un arreglo. Devuelve la referencia a este arreglo (los constructores de las superclases son ignorados).
Field[ ] getDeclaredFields( ) throws SecurityException	Devuelve un objeto **Field** para todos los campos declarados por esta clase y los almacena en un arreglo. Devuelve una referencia a dicho arreglo (los campos heredados se ignoran).
Method[ ] getDeclaredMethods( ) throws SecurityException	Devuelve un objeto **Method** para cada uno de los métodos declarados por esta clase o interfaz y los almacena en un arreglo. Devuelve la referencia de dicho arreglo (los métodos heredados son ignorados).
Field[ ] getFields(String *campoNom*) throws NoSuchMethodException, SecurityException	Devuelve un objeto **Field** que representa el campo especificado por *campoNom* para el objeto que invoca.
Field[ ] getFields( )throws SecurityException	Devuelve un objeto **Field** para todos los campos públicos del objeto que invoca y los almacena en un arreglo. Devuelve la referencia a dicho arreglo.
Class<?>[ ] getInterfaces( )	Cuando se invoca a un objeto, este método devuelve un arreglo de las interfaces implementadas por clase del objeto. Cuando se invoca a una interfaz, este método devuelve un arreglo de interfaces extendidas por la interfaz.

Tabla 16-15 Algunos métodos definidos por la clase **Class** (*continuación*)

Método	Descripción
Method[ ] getMethod(String *m*, Class<?> ...*paramTipos*) throws NoSuchMethodException, SecurityException	Devuelve un objeto de tipo **Method** que representa el método especificado por *m* y que tiene los tipos de parámetros especificados por *paramTipos*.
Method[ ] getMethods( ) throws SecurityException	Obtiene un objeto **Method** para cada método público del objeto que invoca y los almacena en un arreglo. Devuelve la referencia a dicho arreglo.
String getName( )	Devuelve el nombre completo de la clase o la interfaz del objeto que invoca.
ProtectionDomain getProtectionDomain( )	Devuelve el dominio de protección asociado con el objeto que invoca.
Class <? super T>getSuperclass( )	Devuelve la superclase del objeto que invoca. El valor devuelto es **null** si el objeto que invoca es de tipo **Object**.
boolean isInterface( )	Devuelve **verdadero** si el objeto que invoca es una interfaz. De lo contrario devuelve **falso**.
T newInstance( ) throws IllegalAccessException, InstantiationException	Crea una nueva instancia (por ejemplo, un objeto nuevo) que es del mismo tipo que el objeto que invoca. Esto es equivalente a utilizar el operador **new** con el constructor por omisión de la clase. Se devuelve el nuevo objeto creado.
String toString( )	Devuelve una cadena que representa al objeto o interfaz que invoca.

TABLA 16-15 Algunos métodos definidos por la clase **Class** (*continuación*)

Los métodos definidos por **Class** son a menudo útiles en situaciones en que se necesita información de un objeto en tiempo de ejecución. Como lo muestra la Tabla 16-15, la clase proporciona métodos que permiten determinar información adicional sobre una clase concreta, como por ejemplo sus campos, sus métodos y constructores públicos. Además de otras cosas. Esto es importante para el funcionamiento de los Java Beans, el cual se tratará más adelante en este libro.

El siguiente programa ejemplifica el uso del método **getClass( )** (heredado de **Object**) y **getSuperclass( )** (de la clase **Class**):

```
// Ejemplo con información en tiempo de ejecución.

class X {
  int a;
  float b;
}

class Y extends X
  double c;
}

class RTTI {
```

```
  public static void main(String args[]) {
    X x = new X();
    Y y = new y();
    Class <?>clObj;

    clObj = x.getClass(); //obtiene la clase
    System.out.println("x es un objeto del tipo: " +
                       clObj.getName());

    clObj = y.getClass(); //obtiene la clase
    System.out.println("y es un objeto del tipo: " +
                       clObj.getName());

    clObj = clObj.getSuperclass();
    System.out.println("la superclase de y es: " +
                       clObj.getName());
  }
}
```

La salida de este programa es la siguiente:

```
x es un objeto del tipo: X
y es un objeto del tipo: Y
la superclase de y es: X
```

ClassLoader

La clase abstracta **ClassLoader** define cómo se cargan las clases. Una aplicación puede crear subclases que extiendan **ClassLoader**, implementando sus métodos. Esto permite cargar clases de un modo diferente al que normalmente utiliza el intérprete de Java. Sin embargo, esto es algo que no se hace normalmente.

Math

La clase **Math** contiene todas las funciones de punto flotante que se utilizan en geometría y trigonometría, así como varios métodos de propósito general. **Math** define dos constantes **double**: **E**(aproximadamente 2.72) y **PI** (aproximadamente 3.14).

Funciones trascendentes

Los siguientes métodos aceptan parámetros **double** para ángulos en radianes y devuelven el resultado de su función transcendental:

Método	Descripción
static double sin(double *arg*)	Devuelve el seno del ángulo especificado por *arg* en radianes.
static double cos(double *arg*)	Devuelve el coseno del ángulo especificado por *arg* en radianes.
static double tan(double *arg*)	Devuelve la tangente del ángulo especificado por *arg* en radianes.

Los siguientes métodos toman como parámetro el resultado de una función trascendente y devuelven, en radianes, el ángulo que produciría ese resultado. Son las funciones inversas de las anteriores.

Método	Descripción
static double asin(double *arg*)	Devuelve el ángulo cuyo seno viene dado por *arg*.
static double acos(double *arg*)	Devuelve el ángulo cuyo coseno viene dado por *arg*.
static double atan(double *arg*)	Devuelve el ángulo cuya tangente viene dada por *arg*.
static double atan2(double *x*, double *y*)	Devuelve el ángulo cuya tangente es *x/y*.

Los siguientes métodos calculan el seno, coseno y tangente hiperbólicos de un ángulo.

Método	Descripción
static double sinh(double *arg*)	Devuelve el seno hiperbólico de un ángulo especificado por *arg*.
static double cosh(double *arg*)	Devuelve el coseno hiperbólico de un ángulo especificado por *arg*.
static double tanh(double *arg*)	Devuelve la tangente hiperbólica de un ángulo especificado por *arg*.

Funciones exponenciales

Math define los siguientes métodos exponenciales:

Método	Descripción
static double cbrt(double *arg*)	Devuelve la raíz cúbica de *arg*.
static double exp(double *arg*)	Devuelve e elevado a *arg*.
static double expm1(double *arg*)	Devuelve e elevado a *arg*-1.
static double log(double *arg*)	Devuelve el logaritmo natural de *arg*.
static double log10(double *arg*)	Devuelve el logaritmo base 10 de *arg*.
static log1p(double *arg*)	Devuelve el logaritmo natural de *arg*+1.
static double pow(double *y*, double *x*)	Devuelve *y* elevado a *x*; por ejemplo, pow(2.0, 3.0) devuelve 8.0.
static double scalb(double *arg*, int *factor*)	Devuelve val * 2^{factor} (agregado por Java SE 6).
static float scalb(float *arg*, int *factor*)	Devuelve val * 2^{factor} (agregado por Java SE 6).
static double sqrt(double *arg*)	Devuelve la raíz cuadrada de *arg*.

Funciones de redondeo

La clase **Math** define varios métodos que proporcionan distintos tipos de operaciones de redondeo. Las cuales se muestran en la Tabla 16-16. Observe los dos métodos **ulp( )** al final de la tabla. En este contexto, *ulp* representa las siglas en inglés de la frase *"unidades en el último lugar"*. Los métodos *ulp* obtienen el número de unidades entre un valor y el valor superior siguiente. Pueden ser utilizados para evaluar la exactitud de un resultado.

Método	Descripción
static int abs(int *arg*)	Devuelve el valor absoluto de *arg*.
static long abs(long *arg*)	Devuelve el valor absoluto de *arg*.
static float abs(float *arg*)	Devuelve el valor absoluto de *arg*.
static double abs(double *arg*)	Devuelve el valor absoluto de *arg*.
static double ceil(double *arg*)	Devuelve el menor número entero mayor o igual a *arg*.
static double floor(double *arg*)	Devuelve el mayor número entero menor o iguales a *arg*.
static int max(int *x*, int *y*)	Devuelve el máximo de *x* e *y*.
static long max(long *x*, long *y*)	Devuelve el máximo de *x* e *y*.
static float max(float *x*, float *y*)	Devuelve el máximo de *x* e *y*.
static double max(double *x*, double *y*)	Devuelve el máximo de *x* e *y*.
static int min(int *x*, int *y*)	Devuelve el mínimo de *x* e *y*.
static long min(long *x*, long *y*)	Devuelve el mínimo de *x* e *y*.
static float min(float *x*, float *y*)	Devuelve el mínimo de *x* e *y*.
static double min(double *x*, double *y*)	Devuelve el mínimo de *x* e *y*.
static double nextAfter(double *arg*, double *adelante*)	Comenzado con el valor de *arg*, devuelve el siguiente valor en dirección hacia *adelante*. Si *arg* = = *adelante*, entonces *adelante* se devuelve (agregado por Java SE 6).
static float nextAfter(float *arg*, double *adelante*)	Comenzado con el valor de *arg*, devuelve el siguiente valor en dirección hacia *adelante*. Si *arg* = = *adelante*, entonces *adelante* se devuelve (agregado por Java SE 6).
static double nextUp(double *arg*)	Devuelve el siguiente valor en dirección positiva desde *arg* (agregado por Java SE 6).
static float nextUp(float *arg*)	Devuelve el siguiente valor en dirección positiva desde *arg* (agregado por Java SE 6).
static double rint(double *arg*)	Devuelve el entero más cercano en valor a *arg*.
static int round(float *arg*)	Devuelve *arg* redondeando al **int** más cercano.
static long round(double *arg*)	Devuelve *arg* redondeado al **long** más cercano.
static float ulp(float *arg*)	Devuelve el ulp para *arg*.
static double ulp(double *arg*)	Devuelve el ulp para *arg*.

TABLA 16-16 Los métodos de redondeo definidos por la clase **Math**

Otros métodos en la clase Math

Además de los métodos que acabamos de comentar, **Math** define los siguientes métodos:

Método	Descripción
static double copySign(double *arg*, double *signarg*)	Devuelve *arg* con el signo especificado en *signarg*. (agregado por Java SE 6).
static float copySign(float *arg*, double *signarg*)	Devuelve *arg* con el signo especificado en *signarg* (agregado por Java SE 6).
static int getExponent(double *arg*)	Devuelve el exponente base 2 utilizado para la representación binaria de *arg* (agregado por Java SE 6).
static int getExponent(float *arg*)	Devuelve el exponente base 2 utilizado para la representación binaria de *arg* (agregado por Java SE 6).
static double IEEEremainder (double *dividendo*, double *divisor*)	Devuelve el residuo de la división *dividendo/divisor*.
static hypot (double *lado1*, double *lado2*)	Devuelve la longitud de la hipotenusa de un triángulo dada la longitud de dos lados opuestos.
static double random( )	Devuelve un número pseudoaleatorio comprendido entre 0 y 1.
static float signum(double *arg*)	Determina el signo de un valor. Devuelve 0 si *arg* es 0, 1 si *arg* es mayor que 0, y -1 si *arg* es menor que 0.
static float signum(float *arg*)	Determina el signo de un valor. Devuelve 0 si *arg* es 0, 1 si *arg* es mayor que 0, y -1 si *arg* es menor que 0.
static double toRadians(double *a*)	Convierte grados en radianes. El ángulo especificado en *a* debe estar especificado en radianes. Se devuelve el resultado en grados.
static double toDegrees(double *a*)	Convierte radianes en grados. El ángulo especificado en *a* debe estar especificado en grados. Se devuelve el resultado en radianes.

Aquí tenemos un ejemplo del uso de los métodos **toRadians( )** y **toDegrees( )**:

```
// Ejemplo de los métodos toDegrees() y toRadians().
class Angles {
  public static void main(String args[]) {
    double theta = 120.0;

    System.out.println(theta + " grados son" +
                       Math.toRadians(theta) + " radianes.");

    theta =1.312;
    System.out.println(theta + " radianes son " +
                       Math.toDegrees(theta) + " grados.");
  }
}
```

La salida de este programa es:

```
120.0 grados son 2.0943951023931953 radianes.
1.312 radianes son 75.17206272116401 grados.
```

StrictMath

La clase **StrictMath** define un conjunto completo de métodos matemáticos paralelos a los de **Math**. La diferencia es que la versión **StrictMath** garantiza la generación de resultados idénticos en todas las implementaciones de Java, mientras que a los métodos de **Math** se les permite un poco de libertad en su rango de valores para mejorar el rendimiento.

Compiler

La clase **Compiler** permite la creación de entornos de Java en que el código binario de Java se compila en código ejecutable, en lugar de interpretarse. Esta clase no se utiliza en la programación convencional.

Thread, ThreadGroup y Runnable

La interfaz **Runnable** y las clases **Thread** y **ThreadGroup** dan soporte a la programación multihilo. Cada una es examinada a continuación.

NOTA *En el Capítulo 11 se hace una introducción a las técnicas utilizadas para gestionar hilos, implementar la interfaz* ***Runnable*** *y crear programas multihilo.*

La interfaz Runnable

La interfaz **Runnable** debe ser implementada por cualquier clase que inicie un hilo separado de ejecución. **Runnable** sólo define un método abstracto, llamado **run( )**, que es el punto de entrada al hilo. Se define como sigue:

```
abstract void run( )
```

Los hilos creados por el programador deben implementar este método.

Thread

Thread crea un nuevo hilo de ejecución. Define los siguientes constructores comunes:

```
Thread( )
Thread(Runnable objHilo)
Thread(Runnable objHilo, String nomHilo)
Thread(String nomHilo)
Thread(ThreadGroup objGrupo, Runnable objHilo)
Thread(ThreadGroup objGrupo, Runnable objHilo, String nomHilo)
Thread(ThreadGroup objGrupo, String nomHilo)
```

objHilo es una instancia de una clase que implementa la interfaz **Runnable** y define dónde comienza la ejecución de un hilo. El nombre del hilo se especifica en **nomHilo**. Cuando no se proporciona un nombre, la Máquina Virtual de Java crea uno. *objGrupo* especifica el grupo de hilos al que el nuevo hilo pertenecerá. Cuando no se proporciona un grupo de hilos, el nuevo hilo pertenecerá al mismo grupo que su hilo padre.

Thread define las siguientes constantes:

MAX_PRIORITY
MIN_PRIORITY
NORM_PRIORITY

Como sus nombres lo indican, estas constantes especifican las prioridades máxima, mínima y normal de los hilos.

Los métodos definidos por **Thread** se muestran en la Tabla 16-17. En versiones anteriores de Java, la clase **Thread** incluía además a los métodos **stop( )**, **suspend( )** y **resume( )**. Sin embargo, como se explica en el Capítulo 11, éstos se han desechado por ser intrínsecamente inestables. Java también ha descartado al método count **StackFrames( )** debido a que llama a los métodos **suspend( )** y **destroy( )**, lo cual puede causar un bloqueo del tipo conocido como deadlock.

Método	Descripción
static int activeCount( )	Devuelve el número de hilos en el grupo al que pertenece el hilo.
void checkAccess( )	Causa que el gestor de seguridad verifique que el hilo actual pueda acceder y/o cambiar el hilo sobre el que se llama a **checkAccess( )**.
static Thread currentThread( )	Devuelve un objeto **Thread** que encapsula el hilo que llama a este método.
static void dumpStack( )	Muestra la pila de llamadas del hilo.
static int enumerate(Thread *hilos*[ ])	Pone en el arreglo *hilos* copias de todos los objetos **Thread** en el grupo del hilo actual. El método devuelve el número de hilos.
static Map <Thread, StackTrace Element[ ]>getAllStackTraces( )	Devuelve un Map que contiene la pila con el rastro de todos los hilos activos. En el objeto **Map**, cada entrada consiste de una clave, la cual es un objeto **Thread**, y su valor, el cuál es un arreglo de **StackTraceElement**.
ClassLoader getContextClassLoader( )	Devuelve el cargador de clases que se utiliza para cargar clases y recursos para este hilo.
static Thread.UncaughtExceptionHandler getDefaultUncaughExceptionHandler( )	Devuelve el manejador por omisión utilizado para gestionar las excepciones libres.
long getID( )	Devuelve el ID del hilo que invoca.
final String getName( )	Devuelve el nombre del hilo.
final int getPriority( )	Devuelve la prioridad del hilo.
StackTraceElement[ ] getStackTrace( )	Devuelve un arreglo que contiene la pila de rastreo para el hilo que invoca.
Thread.State getState( )	Devuelve el estado del hilo que invoca.
final ThreadGroup getThreadGroup( )	Devuelve el objeto **ThreadGroup** del que el hilo que invoca es miembro.
Thread.UncaughtExceptionHandler getUncaughtExceptionHandler( )	Devuelve el manejador utilizado por el hilo que invoca para gestionar las excepciones libres.
static boolean holdsLock(Object *obj*)	Devuelve **verdadero** si el hilo que invoca posee el candado sobre *obj*. Devuelve **falso** en cualquier otro caso.

TABLA 16-17 Los métodos definidos por la clase **Thread**

Método	Descripción
void interrupt( )	Interrumpe el hilo.
static boolean interrupted( )	Devuelve **verdadero** si el hilo actualmente en ejecución ha sido programado para su interrupción. De lo contrario, devuelve **falso**.
final boolean isAlive( )	Devuelve **verdadero** si el hilo sigue activo. De lo contrario, devuelve **falso**.
final boolean isDaemon( )	Devuelve **verdadero** si el hilo es un hilo demonio. De lo contrario, devuelve **falso**.
boolean isInterrupted( )	Devuelve **verdadero** si el hilo está interrumpido. De lo contrario, devuelve **falso**.
final void join( ) throws InterruptedException	Espera hasta que el hilo termine.
final void join(long *milisegundos*) throws InterruptedException	Espera el número de *milisegundos* especificado a que termine el hilo sobre el que es llamado.
final void join(long *milisegundos*, int *nanosegundos*) throws InterruptedException	Espera el número de *milisegundos* más *nanosegundos* especificados a que termine el hilo sobre el que es llamado.
void run( )	Comienza la ejecución de un hilo.
void setContextClassLoader(ClassLoader *cl*)	Establece al cargador de clases *cl* como el cargador a utilizar por el hilo que invoca.
final void setDaemon(boolean *estado*)	Marca el hilo como un hilo de tipo demonio.
static void setDefaultUncaughtExceptionHandler (Thread.UncaughtExceptionHandler *em*)	Define a *em* como el manejador de excepciones libres por omisión.
final void setName(String *nomHilo*)	Establece el nombre del hilo al indicado en *nomHilo*.
final void setPriority(int *prioridad*)	Establece la prioridad del hilo a la especificada por *prioridad*.
void setUncaughtExceptionHandler (Thread.UncaughtExceptionHandler *em*)	Define a *em* como el manejador de excepciones libres por omisión para el hilo que invoca.
static void sleep(long *milisegundos*) throws InterruptedException	Suspende la ejecución del hilo durante el número de *milisegundos* especificado.
static void sleep(long *milisegundos*, int *nanosegundos*) throws InterruptedException	Suspende la ejecución del hilo durante el número de *milisegundos* más *nanosegundos* especificados.
void start( )	Inicia la ejecución del hilo.
String toString( )	Devuelve la cadena equivalente de un hilo.
static void yield( )	El hilo que invoca cede el CPU a otro hilo.

TABLA 16-17 Los métodos definidos por la clase **Thread** (*continuación*)

ThreadGroup

La clase **ThreadGroup** crea un grupo de hilos. Esta clase define los siguientes dos constructores:

ThreadGroup(String *nomGrupo*)
ThreadGroup(ThreadGroup *objPadre*, String *nomGrupo*)

En ambas formas, *nomGrupo* indica el nombre del grupo de hilos. La primera versión crea un nuevo grupo que tiene al hilo actual como padre. En la segunda forma, el padre se especifica en *objPadre*. Los métodos definidos por **ThreadGroup** se muestran en la Tabla 16-18.

Los grupos de hilos ofrecen una forma cómoda de gestionar conjuntos de hilos como una unidad. Esto es especialmente interesante en situaciones en las que se desea suspender y reanudar simultáneamente una serie de hilos relacionados. Por ejemplo, imagínese un programa en que un conjunto de hilos se utiliza para imprimir un documento, otro para mostrar el documento en pantalla y otro para guardar el documento a un archivo en disco. Si se aborta la impresión, será muy deseable tener un modo de parar todos los hilos relacionados con la impresión.

Método	Descripción
int activeCount( )	Devuelve el número de hilos en el grupo y en todos los grupos de los que este hilo es padre.
int activeGroupCount( )	Devuelve el número de grupos de los que el hilo que invoca es padre.
final void checkAccess( )	Hace que el gestor de seguridad verifique que el hilo que invoca puede acceder y/o cambiar el grupo sobre el que **checkAccess**( ) es llamado.
final void destroy( )	Destruye el grupo de hilos (y todos los grupos hijos) sobre el que se llama.
int enumerate(Thread *grupo*[ ])	Los hilos que pertenecen al *grupo* de hilos que invoca se colocan en el arreglo *grupo*.
int enumerate(Thread *grupo*[ ], boolean *todos*)	Los hilos que pertenecen al *grupo* de hilos que invoca se colocan en el arreglo grupo. Si el parámetro *todos* es **verdadero**, los hilos en todos los subgrupos del hilo también se incluyen en el arreglo *grupo*.
int enumerate(ThreadGroup *grupo*[ ])	Los subgrupos del grupo de hilos que invoca se colocan en el arreglo *grupo*.
int enumerate(ThreadGroup *grupo*[ ], boolean *todos*)	Los subgrupos del grupo de hilos que invoca se colocan en el arreglo *grupo*. Si el parámetro todos es **verdadero**, entonces *todos* los subgrupos de los subgrupos (y así sucesivamente) también se incluyen en el arreglo *grupo*.
final int getMaxPriority( )	Devuelve la prioridad máxima establecida en el grupo.
final String getName( )	Devuelve el nombre del grupo.
final ThreadGroup getParent( )	Devuelve **null** si el objeto **ThreadGroup** que invoca no tiene padre. De lo contrario, devuelve el padre del objeto que invoca.
final void interrupt( )	Llama al método **interrupt( )** de todos los hilos del grupo.

TABLA 16-18 Los métodos definidos por la clase **ThreadGroup**

Método	Descripción
final boolean isDaemon( )	Devuelve **verdadero** si el grupo es un grupo demonio. De lo contrario, devuelve **falso**.
boolean isDestroyed( )	Devuelve **verdadero** si el grupo ha sido destruido. De lo contrario, devuelve **falso**.
void list( )	Muestra información del grupo.
final boolean parentOf(ThreadGroup *grupo*)	Devuelve **verdadero** si el hilo que invoca es el padre de *grupo* (o el *grupo* mismo). De lo contrario, devuelve **falso**.
final void setDaemon(boolean *esDemonio*)	Si esDemonio es **verdadero**, entonces el grupo que invoca se marca como grupo demonio.
final void setMaxPriority(int *prioridad*)	Establece la máxima prioridad del grupo que invoca al valor dado por el parámetro *prioridad*.
String toString( )	Devuelve la cadena equivalente del grupo.
void uncaughtException(Thread *hilo*, Throwable *e*)	Este método es llamado cuando se produce una excepción y ésta no ha sido gestionada.

TABLA 16-18 Los métodos definidos por la clase **ThreadGroup** (*continuación*)

Los grupos de hilos ofrecen esta posibilidad. El siguiente programa, el cual crea dos grupos de hilos de dos hilos cada uno, ilustra este uso:

```
// Uso de grupos de hilos.
class NewThread extends Thread {
  boolean suspendBandera;

  NewThread(String nomHilo, ThreadGroup tgOb){
    super (tgOb, nomHilo);
    System.out.println("Nuevo hilo: " + this);
    suspendBandera = false;
    start(); // Iniciar el hilo
  }

  // Este es el punto de entrada al hilo.
  public void run() {
    try {
      for (int i = 5; i > 0; i--) {
        System.out.println(getName( ) + ": " + i);
        Thread.sleep(l000);
        synchronized(this) {
          while(suspendBandera) {
            wait();
          }
        }
      }
```

```
    } catch (Exception e) {
      System.out.println("Excepción en " + getName());
    }
    System.out.println(getName() + " saliendo.");
  }

  void mysuspend() {
    suspendBandera = true;
  }

  synchronized void myresume() {
    suspendBandera = false;
    notify();
  }
}

class ThreadGroupDemo {
  public static void main(String args[]) {
    ThreadGroup groupA = new ThreadGroup ("Grupo A");
    ThreadGroup groupB = new ThreadGroup ("Grupo B");

    NewThread ob1 = new NewThread ("Uno", groupA);
    NewThread ob2 = new NewThread ("Dos", groupA);
    NewThread ob3 = new NewThread ("Tres", groupB);
    NewThread ob4 = new NewThread ("Cuatro", groupB);

    System.out.println (" \nEsta es la salida de list () : ");
    groupA.list();
    groupB.list();
    System.out.println();

    System.out.println("Suspendiendo Grupo A");
    Thread tga[] = new Thread[groupA.activeCount()];
    groupA.enumerate(tga);   // obtener los hilos en el grupo
    for(int i = 0; i < tga.length; i++) {
      ((NewThread)tga[i]).mysuspend();  // suspender cada hilo
    }

    try {
      Thread.sleep(4000);
    } catch (InterruptedException e) {
      System.out.println("Hilo principal interrumpido.");
    }

    System.out.println("Reanudando Grupo A");
    for(int i = 0; i < tga.length; i++) {
      ((NewThread)tga[i]).myresume(); // reanudar hilos en el grupo
    }

    // Esperar a que los hilos terminen
    try {
      System.out.println("Esperando a que los hilos terminen.");
      ob1.join();
      ob2.join();
      ob3.join();
      ob4.join();
```

```
    } catch (Exception e) {
      System.out.println("Excepción en el hilo principal");
    }

    System.out.println("Hilo principal saliendo.");
  }
}
```

Un ejemplo de salida de este programa se muestra a continuación (la salida obtenida por el usuario puede variar):

```
Nuevo hilo: Thread[Uno,5,Grupo A]
Nuevo hilo: Thread[Dos,5,Grupo A]
Nuevo hilo: Thread[Tres,5,Grupo B]
Nuevo hilo: Thread[Cuatro,5,Grupo B]
Ésta es la salida de list():
java.lang.ThreadGroup[name = Grupo A, maxpri = 10]
  Thread[Uno,5,Grupo A]
  Thread[Dos,5,Grupo A]
java.lang.ThreadGroup[name = Grupo B, maxpri=10]
  Thread[Tres,5,Grupo B]
  Thread[Cuatro,5,Grupo B]
Suspendiendo Grupo A
Tres: 5
Cuatro: 5
Tres: 4
Cuatro: 4
Tres: 3
Cuatro: 3
Tres: 2
Cuatro: 2
Reanudando Grupo A
Esperando a que los hilos terminen.
Uno: 5
Dos: 5
Tres: 1
Cuatro: 1
Uno: 4
Dos: 4
Tres saliendo.
Cuatro saliendo.
Uno: 3
Dos: 3
Uno: 2
Dos: 2
Uno: 1
Dos: 1
Uno saliendo.
```

```
Dos saliendo.
Hilo principal saliendo.
```

Dentro del programa, note que el grupo de hilos A se ha suspendido durante cuatro segundos. Como la salida confirma, esto provoca la pausa de los hilos Uno y Dos, pero los hilos Tres y Cuatro continúan corriendo. Tras los cuatro segundos, los hilos Uno y Dos se reanudan. Obsérvese cómo se suspende y reanuda el grupo de hilos A. Primero se obtienen los hilos del grupo A llamando al método **enumerate( )** sobre el grupo A. Luego, cada hilo se suspende por iteración a lo largo del arreglo resultante. Para reanudar los hilos del grupo A, se recorre de nuevo la lista y se reanuda cada hilo. Una última cosa: este ejemplo utiliza el modo recomendado para suspender y reanudar hilos. No confía en los métodos descartados **suspend( )** y **resume( ).**

ThreadLocal e InheritableThreadLocal

Java define dos clases adicionales relacionadas con hilos en **java.lang**:

- **ThreadLocal**. Se utiliza para crear variables locales de hilo. Cada hilo tendrá su propia copia de una variable local de hilo.
- **InheritableThreadLocal**. Crea variables locales de hilo que pueden ser heredadas.

Package

La clase **Package** encapsula datos de versión asociados a un paquete. La información de versión de un paquete se está volviendo cada vez más importante por la proliferación de paquetes y, porque un programa en Java puede necesitar conocer qué versión de cierto paquete está disponible. Los métodos definidos por la clase **Package** se muestran en la Tabla 16-19. El siguiente programa ilustra la clase **Package** mostrando los paquetes de los que el programa tiene actualmente conocimiento.

```
// Ejemplo de la clase Package
class PkgTest {
  public static void main(String args[]) {
    Package pkgs[];

    pkgs = Package.getPackages();

    for(int i=0; i < pkgs.length; i++)
      System.out.println(
             pkgs[i].getName() +" "+
             pkgs[i].getImplementationTitle() + " " +
             pkgs[i].getImplementationVendor() + " " +
             pkgs[i].getlmplementationVersion()
      );

  }
}
```

Método	Descripción
<A extends Annotation> A getAnnotation(Class <A> *aTipo*)	Devuelve un objeto **Annotation** que contiene la anotación asociada con el *aTipo* para el objeto que invoca.
Annotation[ ]getAnnotations( )	Devuelve todas las anotaciones asociadas con el objeto que invoca en un arreglo de objetos **Annotation**. Devuelve la referencia a dicho arreglo.
Annotation[ ]getDeclaredAnnotations( )	Devuelve un objeto **Annotation** para todas las anotaciones que están declaradas por el objeto que invoca (las anotaciones heredadas se ignoran).
String getImplementationTitle( )	Devuelve el título del paquete que invoca.
String getImplementationVendor( )	Devuelve el nombre del implementador del paquete que invoca.
String getImplementationVersion( )	Devuelve el número de versión del paquete que invoca.
String getName( )	Devuelve el nombre del paquete que invoca.
static Package getPackage(String *nomPaquete*)	Devuelve un objeto **Package** con el nombre especificado por *nomPaquete*.
static Package[ ] getPackages( )	Devuelve todos los paquetes de los que el programa que invoca tiene actualmente conocimiento.
String getSpecificationTitle( )	Devuelve el título de la especificación del paquete que invoca.
String getSpecificationVendor( )	Devuelve el nombre del propietario de la especificación del paquete que invoca.
String getSpecificationVersion( )	Devuelve el número de versión de la especificación del paquete que invoca.
int hashCode( )	Devuelve el código de dispersión del paquete que invoca.
boolean isAnnotationPresent(Class<? extends Annotation> *an*)	Devuelve **verdadero** si la anotación descrita por *an* está asociada con el objeto que invoca. Devuelve **falso** en caso contario.
boolean isCompatible With(String *numVer*) throws NumberFormatException	Devuelve **verdadero** si *numVer* es menor o igual al número de versión del paquete que invoca.
boolean isSealed( )	Devuelve **verdadero** si el paquete que invoca está sellado. Devuelve **falso** en caso contrario.
boolean isSealed(URL *url*)	Devuelve **verdadero** si el paquete que invoca está sellado en relación a *url*. Devuelve **falso** en caso contrario.
String toString( )	Devuelve la cadena equivalente al paquete que invoca.

TABLA 16-19 Los métodos definidos por la clase **Package**

RuntimePermission

La clase **RuntimePermission** se refiere al mecanismo de seguridad de Java y no lo vamos a tratar más aquí.

Throwable

La clase **Throwable** da soporte al sistema de gestión de excepciones de Java, y es la clase de la que se derivan todas las clases de excepción. Esta clase se examina en el Capítulo 10.

SecurityManager

La clase **SecurityManager** es una clase abstracta que puede ser implementada por subclases propias para crear un gestor de seguridad. Generalmente no es necesario implementar un gestor de seguridad propio. Sin embargo, si se desea hacer, es necesario consultar la documentación que viene con el sistema de desarrollo de Java.

StackTraceElement

La clase **StackTraceElement** describe un elemento individual de una *pila de rastreo* cuando ocurre una excepción. Cada elemento de la pila representa un *punto de ejecución*, el cual incluye algunas cosas, tales como, el nombre de la clase, el nombre del método, el nombre de un archivo y el número de la línea del código fuente. Un arreglo de **StackTraceElements** se devuelve mediante el método **getStackTrace( )** de la clase **Throwable**.

La clase **StackTraceElement** tiene un constructor:

StackTraceElement(String *classNom*, String *metNom*, String *nomArch*, int *linea*)

Donde, el nombre de la clase se especifica por *classNom*, el nombre del método se especifica en *metNom*, el nombre del archivo en *nomArch*, y el número de la línea de código se pasa en *línea*. Si no hay un número de línea válido, se utiliza un número negativo para *línea*. Además, un valor de –2 para el argumento *línea*, indica que este elemento de la pila de rastreo hace referencia a un método nativo.

Los métodos soportados por *StackTraceElement* se muestran en la Tabla 16-20. Estos métodos proporcionan acceso programático a la pila de rastreo.

Método	Descripción
boolean equals(Object *obj*)	Devuelve **verdadero** si el **StackTraceElement** que invoca es el mismo que el definido en *obj*. De otra forma, devuelve **falso**.
String getClassName( )	Devuelve el nombre de la clase del punto de ejecución descrito por el objeto **StrackTraceElement** que invoca.
String getFileName( )	Devuelve el nombre del archivo del punto de ejecución descrito por el objeto **StrackTraceElement** que invoca.

TABLA 16-20 Los métodos definidos por la clase **StackTraceElement**

Método	Descripción
int getLineNumber( )	Devuelve el número de la línea del código fuente del punto de ejecución descrito por el objeto **StrackTraceElement** que invoca. En algunos casos, el número de la línea no estará disponible, en tal caso se devuelve un valor negativo.
String getMethodName( )	Devuelve el nombre del método del punto de ejecución descrito por el objeto **StrackTraceElement** que invoca.
int hashCode( )	Devuelve el código de dispersión del objeto **StrackTraceElement** que invoca.
boolean isNativeMethod( )	Devuelve **verdadero** si el objeto **StrackTraceElement** que invoca describe a un método nativo. En cualquier otro caso, devuelve **falso**.
String toString( )	Delvuelve la **cadena** equivalente a la secuencia que invoca.

TABLA 16-20 Los métodos definidos por la clase **StackTraceElement** (*continuación*)

Enum

Tal como se describió en el Capítulo 12, las enumeraciones son una adición reciente al lenguaje Java (recuerde que las enumeraciones son creadas utilizando la palabra clave **enum**). Todas las enumeraciones automáticamente heredan de **Enum**. **Enum** es una clase genérica que se declara como se muestra a continuación:

```
class Enum <E extends Enum<E>>
```

Donde, **E** representa el tipo de la enumeración. **Enum** no tiene constructores públicos.

Enum define varios métodos que están disponibles para ser utilizados en todas las enumeraciones. Estos métodos se describen en la Tabla 16-21.

Método	Descripción
protected final Object clone( ) throws CloneNotSupportedException	Invocar a este método causa que se lance una excepción **CloneNotSupportedException**. Esto previene que las enumeraciones sean clonadas.
final int compareTo(E *e*)	Compara el valor ordinal de dos constantes de la misma enumeración. Devuelve un valor negativo si la constante invocada tiene un valor ordinal menor que el valor ordinal de *e*, cero si los dos valores ordinales son iguales, y un valor positivo si la constante que invoca tiene un valor ordinal mayor que el valor ordinal de *e*.
final boolean equals (Object *obj*)	Devuelve verdadero si *obj* y el objeto que invoca se refieren a la misma constante.
final Class <E>getDeclaringClass( )	Devuelve el tipo de enumeración de la cual la constante que invoca es miembro.
final int hashCode( )	Devuelve el código de dispersión para el objeto que invoca.

TABLA 16-21 Los métodos definidos por la clase **Enum**

Método	Descripción
final String name( )	Devuelve el nombre sin cambio de la constante que invoca.
final int ordinal( )	Devuelve un valor que indica la posición de la constante enumerada en la lista de constantes.
String toString( )	Devuelve el nombre de la constante que invoca. Este nombre podría diferir de la utilizada en la declaración de la enumeración.
static <T extends Enum <T>> T valueOf(Class<T> tipo, String *nom*)	Devuelve la constante asociada con *nom* en la enumeración de tipo especificado por tipo.

TABLA 16-21 Los métodos definidos por la clase **Enum** (*continuación*)

La interfaz CharSequence

La interfaz **CharSequence** define métodos que garantizan acceso de sólo lectura a una secuencia de caracteres. Estos métodos se muestran en la Tabla 16-22. Esta interfaz está implementada por **String**, **StringBuffer** y **StringBuilder**. También es implementada por **CharBuffer**, que forma parte del paquete **java.nio** (descrito posteriormente en este libro).

La interfaz Comparable

Los objetos de clases que implementan a la interfaz **Comparable** se pueden ordenar. En otras palabras, las clases que implementan a la interfaz **Comparable** definen objetos que se pueden comparar de alguna manera que tenga sentido. La interfaz **Comparable** es genérica y se declara como sigue:

```
interface Comparable<T>
```

Donde **T** representa el tipo de objetos que se está comparando.

La interfaz **Comparable** declara un método que se utiliza para determinar lo que Java llama el *orden natural* de instancias de una clase. La firma del método se muestra aquí:

```
int compareTo(T obj)
```

Método	Descripción
char charAt(int *idx*)	Devuelve el carácter en el índice especificado por *idx*.
int length( )	Devuelve el número de caracteres en la secuencia que invoca.
CharSequence subSequence(int *inicioidx*, int *finidx*)	Devuelve un subconjunto de la secuencia que invoca comenzando en *inicioidx* y terminando en *finidx*–1.
String toString( )	Devuelve la **cadena** equivalente a la secuencia que invoca.

TABLA 16-22 Los métodos definidos por la clase **CharSequence**

Este método compara el objeto que invoca con *obj*. Devuelve 0 si los valores son iguales, un valor negativo si el objeto que invoca tiene un valor menor. De lo contrario, devuelve un valor positivo.

Esta interfaz está implementada por varias de las clases ya vistas en este libro. Específicamente, las clases **Byte**, **Character**, **Double**, **Float**, **Long**, **Short**, **String** e **Integer** definen un método **compareTo( )**. Además, como explica el próximo capítulo, los objetos que implementan esta interfaz se pueden utilizar en diferentes colecciones.

La interfaz Appendable

Los objetos de una clase que implementan la interfaz **Appeandable** pueden tener un carácter o secuencias de caracteres adjuntos. La interfaz **Appeandable** define estos tres métodos:

```
Appeandable append(char ch) throws IOException
Appeandable append(CharSequence chars) throws IOException
Appeandable append(CharSequence chars, int comienzo, int final) throws IOException
```

En la primer forma, el carácter *ch* es añadido al objeto que invoca. En la segunda forma, la secuencia de caracteres *chars* se añade al objeto que invoca. La tercera forma permite indicar una porción (especificada por *inicio* y *final*) de la secuencia especificada por *chars*. En todos los casos se devuelve una referencia al objeto que invoca.

La interfaz Iterable

La interfaz **Iterable** debe ser implementada por cualquier clase cuyos objetos vayan a ser utilizados en un ciclo estilo for-each. En otras palabras, para que un objeto sea utilizado dentro de un ciclo estilo for-each, su clase debe implementar a la interfaz **Iterable**. La interfaz **Iterable** es una interfaz genérica que tiene esta declaración:

```
interface Iterable<T>
```

Donde, **T** es el tipo del objeto que será iterado. La interfaz define un método, **iterator( )**, el cuál se muestra a continuación:

```
Iterator<T>iterator( )
```

Este método devuelve un iterador para los elementos contenidos en el objeto que invoca.

NOTA *Los iteradores se describen a detalle en el Capítulo 17.*

La interfaz Readable

La interfaz **Readable** indica que un objeto puede ser utilizado como una fuente de caracteres. Define un método llamado **read( )**, el cual se muestra aquí:

```
int read(CharBuffer buf) throws IOException
```

Este método lee caracteres dentro de *buf*. Devuelve el número de caracteres leídos, o –1 si encuentra EOF.

Los subpaquetes de java.lang

Java define varios subpaquetes:

- java.lang.annotation
- java.lang.instrument
- java.lang.management
- java.lang.ref
- java.lang.reflect

A continuación una breve descripción de cada uno:

java.lang.annotation

La característica de las anotaciones de Java está soportada por **java.lang.annotation**, que define la interfaz **Annotation**, y las enumeraciones **ElementType** y **RetentionPolicy**. Las anotaciones se describen en el Capítulo 12.

java.lang.instrument

java.lang.instrument define características que pueden ser utilizadas para agregar instrumentación a varios aspectos de la ejecución de un programa. Define las interfaces **Instrumentation** y **ClassFileTransformer** y la clase **ClassDefinition**.

java.lang.management

El paquete **java.lang.management** provee soporte para la administración de la JVM y el ambiente de ejecución. Utilizando las características en **java.lang.management**, se pueden observar y administrar varios aspectos de un programa en ejecución.

java.lang.ref

Ya hemos visto que las facilidades para la recolección de basura proporcionadas por Java determinan cuándo no existen referencias a un objeto. Se supone entonces que el objeto no se volverá a necesitar y su memoria se recicla. Las clases en el paquete **java.lang.ref**, facilitan un control más flexible sobre el proceso de recolección de basura. Por ejemplo, supóngase que un programa ha creado numerosos objetos que se querrán volver a utilizar más adelante. Se puede seguir manteniendo referencias a esos objetos, pero eso puede requerir mucha memoria.

En lugar de eso, se pueden definir referencias "suaves" a esos objetos. Un objeto que es "accesible suavemente" puede ser reciclado por el recolector de basura si baja la memoria disponible. En ese caso, el recolector de basura asigna **null** a las referencias "suaves" a ese objeto. En caso contrario, el recolector de basura guarda al objeto para su posible uso futuro.

Un programador tiene la posibilidad de determinar si un objeto "suavemente accesible" ha sido reciclado o no. Si ha sido reciclado, se puede volver a crear. De lo contrario, el objeto sigue

disponible para su reutilización. También se pueden crear referencias "débiles" y "fantasmas" a objetos. El estudio de estas y otras características del paquete **java.lang.ref** queda fuera del ámbito de este libro.

java.lang.reflect

La *reflexión* es la capacidad de un programa para analizarse a sí mismo. El paquete **java.lang.reflect** proporciona la capacidad de obtener información sobre los campos, constructores, métodos y modificadores de una clase. Entre otras cosas, se necesita esta información para construir herramientas de software que permitan trabajar con componentes de Java Beans. Las herramientas utilizan la reflexión para determinar dinámicamente las características de un componente. La reflexión fue introducida en el Capítulo 12 y es también examinado en el Capítulo 27.

El paquete **java.lang.reflect** define varias clases, incluyendo **Method**, **Field** y **Constructor**. También define varias interfaces, incluyendo **AnnotatedElement**, **Member** y **Type**. Adicionalmente, el paquete **java.lang.reflect** incluye una clase **Array** que permite la creación y acceso a arreglos dinámicamente.

CAPÍTULO 17

java.util parte 1: colecciones

En este capítulo inicia nuestro análisis del paquete **java.util**. Este importante paquete contiene una gran variedad de clases e interfaces que proporcionan diversas funcionalidades. Por ejemplo, **java.util** tiene clases para generar números pseudo aleatorios, gestionar fechas y horas, observar eventos, manipular conjuntos de bits, separar cadenas en partes y manipular el formato de datos. El paquete **java.util** contiene además uno de los subsistemas más poderosos de Java: *la estructura de colecciones*. La estructura de colecciones es una sofisticada jerarquía de interfaces y clases que proporcionan la tecnología necesaria para la administración de grupos de objetos. Esto amerita un estudio cuidadoso por parte de cualquier programador.

Debido a la gran cantidad de funcionalidades que proporciona **java.util** este paquete es considerablemente grande. Ésta es una lista de las clases disponibles en el paquete **java.util**:

AbstractCollection	EventObject	Random
AbstractList	FormattableFlags	ResourceBundle
AbstractMap	Formatter	Scanner
AbstractQueue	GregorianCalendar	ServiceLoader (añadida en Java SE 6)
AbstractSequentialList	HashMap	SimpleTimeZone
AbstractSet	HashSet	Stack
AbstractDequeue (añadida en Java SE 6)	Hashtable	StringTokenizer
ArrayList	IdentityHashMap	Timer
Arrays	LinkedHashMap	TimerTask
BitSet	LinkedHashSet	TimeZone
Calendar	LinkedList	TreeMap
Collections	ListResourceBundle	TreeSet
Currency	Locale	UUID
Date	Observable	Vector
Dictionary	PriorityQueue	WeakHashMap
EnumMap	Properties	
EnumSet	PropertyPermission	
EventListenerProxy	PropertyResourceBundle	

java.util define las siguientes interfaces:

Collection	List	Queue
Comparator	ListIterator	RandomAccess
Dequeue (añadida en Java SE 6)	Map	Set
Enumeration	Map.Entry	SortedMap
EventListener	NavigableMap (añadida en Java SE 6)	SortedSet
Formattable	NavigableSet (añadida en Java SE 6)	
Iterator	Observer	

Este capítulo examina los miembros de **java.util** que son parte de la estructura de colecciones y el Capítulo 18 analiza sus otras clases e interfaces.

Introducción a las colecciones

La estructura de colecciones de Java estandariza el modo en que los programas trabajan con grupos de objetos. Las colecciones no formaban parte de la versión original de Java, fueron añadidas en J2SE 1.2. Anterior a la existencia de la estructura de colecciones, Java proporcionaba clases ad hoc, como **Dictionary**, **Vector**, **Stack** y **Properties** para almacenar y manipular grupos de objetos. Aunque estas clases eran bastante útiles, les faltaba un contexto central que las unificara. Así por ejemplo, el modo de usar **Vector** era distinto del modo de usar **Properties**. Esta aproximación anterior no estaba diseñada para ser fácilmente extensible o adaptable. Las colecciones son una respuesta a este (y otros) problemas.

La estructura de colecciones se diseñó para conseguir diferentes objetivos. Primero, debía proporcionar un alto rendimiento. Las implementaciones de las colecciones fundamentales (arreglos dinámicos, listas enlazadas, árboles y tablas de dispersión) son altamente eficientes. Rara vez o nunca es necesario escribir manualmente código para alguna de estas "máquinas de datos". Segundo, la estructura de colecciones debía permitir que diferentes tipos de colecciones trabajaran en forma similar y con un alto grado de interoperabilidad. Tercero, debía ser fácil extender y/o adaptar una colección. Para este fin, toda la estructura de colecciones se ha diseñado alrededor de un conjunto de interfaces estándar y varias implementaciones estándar de estas interfaces (como **LinkedList**, **HashSet** y **TreeSet**), Se proporcionan listas para usarse. También se puede implementar una colección propia, si se desea. Varias implementaciones de propósito especial han sido creadas para mayor comodidad del programador, y algunas implementaciones parciales se proporcionan para facilitar la creación de colecciones propias. Finalmente, se han añadido mecanismos que permiten la integración de arreglos estándar dentro de la estructura de colecciones.

Los *algoritmos* son otra parte importante del mecanismo de colecciones. Los algoritmos operan sobre colecciones, se definen como métodos estáticos dentro de la clase **Collections** y están disponibles para todas las colecciones. No es necesario que cada clase en la estructura de colecciones implemente versiones propias. Los algoritmos aportan medios estándares para la manipulación de colecciones.

Otro elemento asociado con la estructura de colecciones es la interfaz **Iterator**. Un *iterador* proporciona un medio de propósito general y estandarizado para acceder a los elementos dentro de una colección, uno por uno. Así, un iterador proporciona un medio para *enumerar* los contenidos de una colección. Debido a que toda colección implementa la interfaz **Iterator**, los elementos de cualquier clase de la estructura de colecciones son accesibles mediante los métodos definidos por **Iterator**. Así, pequeños cambios, el código

para hacer un ciclo por los elementos de un conjunto también se puede utilizar para hacer un ciclo por los elementos de una lista.

Además de las colecciones, la estructura define varias interfaces de mapeo y clases. Los *mapas* almacenan pares clave/valor. Aunque los mapas no son "colecciones" propiamente dichas, están totalmente integrados a la estructura de colecciones. Es posible obtener una vista como colección de un mapa. Esta vista contiene los elementos del mapa almacenados en una colección. Así se pueden procesar los contenidos de un mapa como si fuera una *colección* si se desea.

El mecanismo de colecciones se ha adaptado a algunas de las clases originalmente definidas por **java.util** para que también se pudieran integrar en el nuevo sistema. Es importante entender que aunque la adición de colecciones alteró la arquitectura de muchas de las clases de utilería, no causó el descarte de ninguna. Las colecciones simplemente proporcionan una mejor manera de hacer varias cosas.

Nota *Para los lectores familiarizados con C++ les ayudará saber que la tecnología de colecciones de Java es parecida en su espíritu a la tecnología de la Biblioteca de Plantillas Estándar (STL por sus siglas en inglés) definida por C++. Lo que C++ llama contenedor, Java lo llama colección. Sin embargo existen diferencias importantes entre la estructura de colecciones de Java y STL.*

Cambios recientes en las colecciones

Recientemente, la estructura de colecciones sufrió cambios fundamentales que incrementaron significativamente su poder y modernizaron su uso. Los cambios fueron ocasionados por la adición, en el JDK 5, de tipos parametrizados, autoboxing/unboxing y ciclos estilo for-each. Aunque revisaremos estos elementos a lo largo de este capítulo, veremos una breve introducción a continuación.

Los tipos parametrizados se aplican a las colecciones

La adición de tipos parametrizados causó un cambio significativo a la estructura de colecciones. Todas las colecciones soportan en la actualidad tipos parametrizados y muchos de los métodos que trabajan con colecciones utilizan parámetros con tipos parametrizados. Agregar tipos parametrizados afectó todas las partes de la estructura de colecciones.

Los tipos parametrizados agregaron una cualidad hasta ese momento ausente de las colecciones: seguridad en el manejo de tipos. Antes de los tipos parametrizados todas las colecciones almacenaban referencias a instancias de la clase **Object**, lo cual significa que cualquier colección podía almacenar cualquier tipo de objeto. Por ello, era posible de manera accidental almacenar elementos de tipos no compatibles en una colección. Lo cual a su vez podría causar un error, en tiempo de ejecución, de incompatibilidad de tipos. Utilizando tipos parametrizados es posible declarar explícitamente el tipo de dato a ser almacenado y por ende eliminar los errores de incompatibilidad de tipos.

Aunque anexar tipos parametrizados cambió la declaración de la mayoría de las clases e interfaces, así como de una gran cantidad de métodos, en su conjunto la estructura de colecciones continua funcionando de la misma forma que antes. No obstante, si el lector está familiarizado con la versión previa de la estructura de colecciones (sin tipos parametrizados) quizá encuentre la nueva sintaxis un poco intimidatoria. No debemos preocuparnos, con el tiempo nos acostumbraremos a la sintaxis de tipos parametrizados.

Finalmente debemos mencionar que para aprovechar las ventajas que los tipos parametrizados aportan a las colecciones, será necesario reescribir código que programas viejos. Esto es muy importante debido a que el códig enerará mensajes de advertencia

cuando sea compilado con una versión actual del compilador de Java. Para eliminar estos mensajes será necesario añadir parametrización de tipos a todo código relacionado con el manejo de colecciones.

El autoboxing facilita el uso de tipos primitivos

El uso de autoboxing y unboxing facilita el almacenamiento de tipos primitivos en colecciones. Como se verá más adelante una colección sólo puede almacenar referencias, no tipos primitivos. En el pasado, si se deseaba almacenar un tipo primitivo, como un **entero**, en una colección, era necesario manualmente encerrar el valor primitivo en un objeto que lo envolviera.

Cuando se requería utilizar el valor era necesario manualmente sacarlo del objeto que lo envolvía. Con el uso de autoboxing y unboxing, Java puede realizar automáticamente la envoltura y desenvoltura necesarias cuando se almacenan y recuperan tipos primitivos, ya no es necesario realizar manualmente estas operaciones.

El ciclo estilo for-each

Otra de las mejoras aplicables a todas las clases que representan colecciones en la estructura de colecciones es que ahora implementan la interfaz **Iterable**, lo cual significa que todas pueden ser recorridas linealmente utilizando un ciclo estilo **for-each**. En el pasado, recorrer linealmente una colección requería del uso de un iterador (los iteradotes se describen más adelante en este capítulo), con el cual el programador construía manualmente un ciclo. Aunque los iteradores aún son necesarios para algunos propósitos, en muchos casos, los ciclos basados en iteradores pueden ser reemplazados por ciclos **for**.

Las interfaces de la estructura de colecciones

La estructura de colecciones define diversas interfaces. Esta sección da una visión general de cada una de ellas. Es necesario comenzar con las interfaces de colección porque determinan la naturaleza fundamental de las clases de colección. Dicho de otro modo, las clases concretas simplemente proporcionan diferentes implementaciones de las interfaces estándar. Las interfaces que son base fundamental de las colecciones se resumen en la siguiente tabla:

Interfaz	Descripción
Collection	Permite trabajar con grupos de objetos; está en la cima de la jerarquía de colecciones.
Dequeue	Extiende a la interfaz **Queue** para manejar una fila doble. (Añadida en Java SE 6).
List	Extiende la interfaz **Collection** para manejar secuencias (listas de objetos).
NavigableSet	Extiende la interfaz **SortedSet** para manejar recuperación de información con base a búsquedas de coincidencias próximas (Añadido por Java SE 6).
Queue	Extiende la interfaz **Collection** para manejar un tipo especial de lista donde los elementos son eliminados sólo desde el inicio de la lista.
Set	Extiende la interfaz **Collection** para manejar conjuntos, los cuales deben contener elementos únicos.
SortedSet	Extiende la interfaz **Set** para manejar conjuntos ordenados.

Además de las interfaces de colección, las colecciones también utilizan las interfaces **Comparator**, **RandomAccess**, **Iterator** y **ListIterator**, que se describen en profundidad más adelante en este capítulo. Brevemente, **Comparator** define cómo se comparan dos objetos;

Iterator y **ListIterator** enumeran los objetos dentro de una colección. Implementando **RandomAccess**, una lista indica que soporta el acceso aleatorio a sus elementos.

Para proporcionar la máxima flexibilidad en su uso, las interfaces de colección permiten que algunos métodos sean opcionales. Los métodos opcionales permiten modificar los contenidos de una colección. Las colecciones que soportan estos métodos se llaman *modificables*. Las que no permiten que sus contenidos cambien se llaman *no modificables*. Si se intenta utilizar uno de esos métodos en una colección no modificable, se produce una excepción del tipo **UnsupportedOperationException**. Todas las colecciones incorporadas en la estructura de colecciones son modificables.

Las siguientes secciones examinan las interfaces de colección.

La interfaz collection

La interfaz **Collection** es la base sobre la que se construye la estructura de colecciones. Toda clase que defina una colección debe implementar esta interfaz. La interfaz **Collection** es una interfaz genérica declarada como:

```
interface Collection<E>
```

Aquí **E** especifica el tipo de objeto que la colección almacenará. La interfaz **Collection** extiende a la interfaz **Iterable**. Esto significa que todas las colecciones pueden ser recorridas utilizando un ciclo estilo **for-each**. Recuerde que sólo las clases que implementan la interfaz **Iterable** pueden ser recorridas con un ciclo **for**.

La interfaz **Collection** declara los métodos medulares que tendrán todas las colecciones. Estos métodos se resumen en la Tabla 17-1. Dado que todas las colecciones implementan la interfaz **Collection**, es necesaria cierta familiaridad con sus métodos para comprender claramente la Estructura de Colecciones de Java. Varios de estos métodos pueden producir una excepción de tipo **UnsupportedOperationException**. Como se ha explicado, esto ocurre si una colección no puede ser modificada. Una excepción de tipo **ClassCastException** se genera cuando un objeto es incompatible con otro, como por ejemplo cuando se intenta añadir a una colección un objeto incompatible. Una excepción de tipo **NullPointerException** se genera si se intenta almacenar un objeto **null** en una colección que no soporte almacenar elementos con este valor. Una excepción de tipo **IllegalStateException** se genera cuando se intenta agregar un elemento a una colección de tamaño predefinido y ésta se encuentra llena.

Los objetos se añaden a una colección llamando al método **add( ).** Nótese que el método **add( )** toma un argumento de tipo E, lo que significa que los objetos que se añadan a la colección deben ser compatibles con el tipo de dato esperado por la colección. Es posible añadir todos los elementos contenidos en una colección a otra llamando al método **addAll( ).**

Se puede quitar un objeto de la colección utilizando el método **remove( ).** Para quitar un grupo de objetos se llama al método **removeAll( )**. Es posible quitar todos los elementos excepto los de un grupo específico llamando al método **retainAll( )**. Para vaciar una colección se utiliza el método **clear( )**.

Se puede determinar si una colección contiene un objeto específico llamando al método **contains( )**. Para determinar si una colección contiene todos los elementos de otra, se utiliza el método **containsAll( )**. Se puede saber si una colección está vacía llamando al método **is Empty( )**. El número de elementos contenidos actualmente en una colección se obtiene llamando al método **size( )**.

Los métodos **toArray( )** devuelve un arreglo que contiene los elementos almacenados en la colección que invoca. El primero regresa un arreglo de objetos tipo **Object**. El segundo regresa

un arreglo de elementos que tienen el mismo tipo que el arreglo especificado como parámetro. Normalmente esta segunda forma es más apropiada debido a que regresa el arreglo con el tipo deseado. Estos métodos son más importantes de lo que podría parecer a primera vista. A menudo, procesar el contenido de una colección utilizando la sintaxis del manejo de arreglos es benéfico. Proporcionando un camino de conexión entre las colecciones y los arreglos, se puede tener lo mejor de ambos mundos.

Se puede comparar si dos colecciones son iguales llamando al método **equals( )**. El significado preciso de "igualdad" puede diferir entre colecciones. Por ejemplo, se puede implementar **equals( )** de modo que compare los valores de los elementos almacenados en una colección. Alternativamente, **equals( )** puede comparar las referencias a esos elementos.

Otro método muy importante es **iterator( )**, el cual devuelve un iterador a una colección. Los iteradores son utilizados frecuentemente cuando se trabaja con colecciones.

La interfaz List

La interfaz **List** extiende la interfaz **Collection** y declara el comportamiento de una colección que almacena una sucesión de elementos. Se puede insertar o acceder a elementos por su posición en la lista, utilizando una indexación con inicio en cero.

Método	Descripción
boolean add(E *obj*)	Añade *obj* a la colección que invoca. Devuelve **true** si *obj* fue añadido a la colección. Devuelve **false** si *obj* ya es un miembro de la colección y la colección no admite duplicados.
boolean addAll(Collection <? extends E> *c*)	Añade todos los elementos de *c* a la colección que invoca. Devuelve **true** si la operación tuvo éxito (esto es, los elementos fueron añadidos). De lo contrario, devuelve **false**.
void clear( )	Quita todos los elementos de la colección que invoca.
boolean contains(Object *obj*)	Devuelve **true** si *obj* es un elemento de la colección que invoca. De lo contrario, devuelve **false**.
boolean containsAll(Collection <?> *c*)	Devuelve **true** si la colección que invoca contiene todos los elementos de *c*. De lo contrario, devuelve **false**.
boolean equals(Object *obj*)	Devuelve **true** si la colección que invoca y *obj* son iguales. De lo contrario, devuelve **false**.
int hashCode( )	Devuelve el código de dispersión de la colección que invoca.
boolean isEmpty( )	Devuelve **true** si la colección que invoca está vacía. De lo contrario, devuelve **false**.
Iterator <E> iterator( )	Devuelve un iterador para la colección que invoca.
boolean remove(Object *obj*)	Quita una instancia de *obj* de la colección que invoca. Devuelve **true** si el elemento fue quitado. De lo contrario, devuelve **false**.
boolean removeAll(Collection <?> *c*)	Quita de la colección que invoca todos los elementos de *c*. Devuelve **true** si la colección ha cambiado (esto es, si se han quitado elementos). De lo contrario, devuelve **false**.
int size( )	Devuelve el número de elementos contenidos en la colección que invoca.
Object[ ] toArray( )	Devuelve un arreglo que contiene todos los elementos almacenados en la colección que invoca. Los elementos del arreglo son copias de los elementos de la colección.

TABLA 17-1 Los métodos definidos por **Collection**

Método	Descripción
<T> T[ ] toArray (T *a* [ ])	Devuelve un arreglo que contiene los elementos de la colección que invoca. Los elementos del arreglo son copias de los elementos de la colección. Si el tamaño de *a* es igual al número de elementos, estos se devuelven en *a*. Si el tamaño de *a* es menor que el número de elementos, se asigna memoria para un nuevo arreglo del tamaño necesario y ese nuevo arreglo es devuelto. Si el tamaño de *a* es mayor que el número de elementos, al elemento de *a* siguiente al último elemento de la colección se le asigna **null**. Si algún elemento de la colección es de un tipo que no es un subtipo de *a* se produce una excepción de tipo **ArrayStoreException**.

TABLA 17-1 Los métodos definidos por **Collection** *(continuación)*

Una lista puede contener elementos duplicados. **List** es una interfaz genérica declarada como:

```
interface List<E>
```

Donde **E** especifica el tipo de los objetos que la lista debe contener.

Además de los métodos definidos por **Collection**, **List** define algunos propios, los cuales se resumen en la Tabla 17-2. Nótese de nuevo que varios de estos métodos producirán una excepción de tipo **UnsupportedOperationException** si la lista no se puede modificar, y una excepción **ClassCastException** cuando un objeto es incompatible con otro, como cuando se intente añadir un objeto incompatible a la lista. Varios métodos generan una excepción de tipo **IndexOutOf**

Método	Descripción
void add(int *indice*, E *obj*)	Inserta *obj* en la lista que invoca en la posición correspondiente al índice pasado en *indice*. Los elementos preexistentes en esa posición y más allá del punto de inserción se corren hacia arriba. Así no se sobrescribe ningún elemento.
boolean addAll(int *indice*, Collection <? Extends E> *c*)	Inserta todos los elementos de *c* en la lista que invoca en el índice especificado. Los elementos preexistentes en la posición dada por *indice* y más allá se desplazan hacia arriba para hacer sitio a los nuevos elementos sin sobrescribir ningún elemento. Devuelve **true** si la lista que invoca cambia, y **false**, si no.
E get(int *indice*)	Devuelve el objeto almacenado en el índice especificado dentro de la colección que invoca.
int indexOf(Object *obj*)	Devuelve el índice de la primera instancia de *obj* en la lista que invoca. Si *obj* no es un elemento de la lista, se devuelve -1.
ListIterator<E> listIterator( )	Devuelve un iterador al inicio de la lista que invoca.
ListIterator<E> listIterator(int *indice*)	Devuelve un iterador a la lista que invoca que comienza en el índice especificado.
E remove(int *indice*)	Quita de la lista que invoca el elemento en la posición dada por *indice* y devuelve el elemento borrado. La lista resultante es compactada. Esto es, los índices de los elementos subsiguientes disminuyen en uno.
E set(int *indice*, E *obj*)	Asigna *obj* a la posición especificada por *indice* dentro de la lista que invoca.
List<E> subList(int *inicio*, int *fin*)	Devuelve una lista que incluye los elementos desde *inicio* hasta *fin*-1 en la lista que invoca. Los elementos en la lista devuelta también son referenciados por el objeto que invoca.

TABLA 17-2 Los métodos definidos por **List**

BoundException si se utiliza con ellos un índice inválido. Una excepción de tipo **NullPointer Exception** se genera si se intenta almacenar el un objeto **null** y elementos de valor **null** no están permitidos en la lista. Una excepción de tipo **IllegalArgumentException** se genera si se proporciona un argumento no válido.

A las versiones de **add( )** y **addAll( )** definidas por **Collection**, **List** añade los métodos **add(int,E)** y **addAll(int,Collection)**. Estos métodos insertan elementos en la posición especificada. Además, las semánticas de los métodos **add(E)** y **addAll(Collection)** definidos en **Collection** son cambiadas por **List** para que añadan los elementos al final de la lista.

Para obtener el objeto almacenado en una posición específica, se llama a **get( )** con el índice del objeto. Para asignar un valor a un elemento de la lista, se llama a **set( )**, especificando el índice del objeto que se ha de cambiar. Para encontrar el índice de un objeto, se utiliza **indexOf( )** o **lastIndexOf( )**.

Se puede obtener una sublista de una lista llamando al método **subList( )**, especificando los índices inicial y final de la sublista.

La interfaz Set

La interfaz **Set** define un conjunto. Extiende la interfaz **Collection** y declara el comportamiento de una colección que no permite elementos duplicados. Por tanto, el método **add( )** devuelve **false** si se intenta añadir elementos duplicados en el conjunto.

No define ningún método adicional propio. **Set** es una interfaz genérica declarada como:

```
interface Set<E>
```

Donde, **E** especifica el tipo de objetos que el conjunto almacenará

La interfaz SortedSet

La interfaz **SortedSet** extiende a la interfaz **Set** y declara el comportamiento de un conjunto ordenado en orden ascendiente. **SortedSet** es una interfaz genérica declarada como:

```
Interface SortedSet<E>
```

Donde, **E** especifica el tipo de objetos que el conjunto almacenará.

Además de los métodos definidos por **Set**, la interfaz **SortedSet** declara los métodos mostrados en la Tabla 17-3. Varios métodos producen una excepción **NoSuchElementException** cuando no hay ningún elemento en el conjunto que invoca. Se produce una excepción **ClassCastException** cuando un objeto es incompatible con los elementos de un conjunto. Se produce una excepción **NullPointerException** si se intenta utilizar un objeto **null** y **null** no está permitido en el conjunto. Se genera una excepción **IllegalArgumentException** si un argumento no permitido es dado a un método.

SortedSet define varios métodos que hacen el procesado de conjuntos más cómodo. Para obtener el primer objeto en el conjunto, se llama al método **first( )**. Para obtener el último elemento se llama al método **last( )**. Para obtener un subconjunto de un **SortedSet** se llama al método **subSet( )**. Para obtener el subconjunto que comienza con el primer elemento del conjunto, se utiliza el método **headSet( )**. Si se quiere obtener el subconjunto que termina el conjunto, se utiliza el método **tailSet( )**.

La interfaz NavigableSet

La interfaz **NavigableSet** fue añadida en Java SE 6. Esta interfaz extiende **SortedSet** y declara el comportamiento de una colección que soporta la recuperación de elementos basada en la

correspondencia más cercana entre un valor o valores. **NavigableSet** es una interfaz genérica declarada como:

```
interface NavigableSet<E>
```

Donde, **E** especifica el tipo de los objetos que el conjunto almacenará. Además a los métodos heredados de **SortedSet**, la interfaz **NavigableSet** añade los métodos listados en la Tabla 17-4.

Método	Descripción
Comparator <? super E> comparator( )	Devuelve el comparador del conjunto ordenado que invoca. Si para este conjunto se utiliza la ordenación natural, se devuelve **null**.
E first( )	Devuelve el primer elemento en el conjunto ordenado que invoca.
SortedSet<E> headSet(E fin)	Devuelve un **SortedSet** que contiene los elementos menores que el valor del parametro fin contenidos en el conjunto ordenado que invoca. Los elementos en el conjunto ordenado devuelto también son referenciados por el conjunto ordenado que invoca.
E last( )	Devuelve el último elemento en el conjunto ordenado que invoca.
SortedSet<E> subSet(E inicio, E fin)	Devuelve un **SortedSet** que incluye los elementos entre inicio y fin-l. Los elementos en la colección devuelta también son referenciados por el objeto que invoca.
SortedSet<E> tailSet(E inicio)	Devuelve un **SortedSet** que contiene los elementos mayores o iguales que inicio contenidos en el conjunto ordenado. Los elementos en el conjunto devuelto también son referenciados por el objeto que invoca.
E ceiling (E *obj*)	Busca en el conjunto al elemento *e* más pequeño tal que *e* >=*obj*. Si encuentra un elemento con estas características lo devuelve en caso contrario devuelve **null**.

TABLA 17-3 Los métodos definidos por **SortedSet**

Método	Descripción
Iterator<E> descendingIterator( )	Devuelve un iterador que realiza un recorrido del elemento más grande al más pequeño. En otras palabras, este método devuelve un iterador inverso.
NavigableSet<E> descendingSet( )	Devuelve un **NavigableSet** que es el inverso del objeto que invoca. El conjunto resultante es una referencia al conjunto que invoca.
E floor (E *obj*)	Busca en el conjunto al elemento *e* más grande tal que *e*<=*obj*. Si encuentra un elemento con estas características lo devuelve en caso contrario devuelve **null**.
NavigableSet<E> headSet(E *upperBound*, boolean *i*)	Devuelve un **NavigableSet** que incluye todos los elementos del conjunto que invoca que son menores que *upperBound*. Si *i* es **true** entonces un elemento igual a *upperBound* también se incluiría en el resultado. El conjunto resultante es una referencia al conjunto que invoca.
E higher(E *obj*)	Busca en el conjunto el elemento *e* más grande tal que *e*>*obj*. Si encuentra un elemento con estas características lo devuelve en caso contrario devuelve **null**.
E lower(E *obj*)	Busca en el conjunto el elemento *e* más grande tal que *e*<*obj*. Si encuentra un elemento con estas características lo devuelve en caso contrario devuelve **null**.

TABLA 17-4 Los métodos definidos por **NavigableSet**

Método	Descripción
E pollFirst( )	Devuelve el primer elemento y lo elimina del conjunto que invoca. Debido a que el conjunto está ordenado, el elemento eliminado es el elemento con el valor menor. Devuelve **null** si el conjunto está vacío.
E pollLast( )	Devuelve el último elemento y lo elimina del conjunto que invoca. Debido a que el conjunto está ordenado, el elemento eliminado es el elemento con el valor mayor. Devuelve **null** si el conjunto está vacío.
NavigableSet<E> subSet(E *lowerBound*, boolean *low*, E *upperBound*, boolean *high*)	Devuelve un **NavigableSet** que incluye todos los elementos del conjunto que invoca que son mayores que *lowerBound* y menores que *upperBound*. Si *low* es **true**, entonces un elemento igual a *lowerBound* sería incluido en el resultado. Si *high* es **true** entonces un elemento igual a *upperBound* sería incluido en el resultado. El conjunto resultante es una referencia al conjunto que invoca.
NavigableSet<E> tailSet (E *lowerBound*, boolean *i*)	Devuelve un **NavigableSet** que incluye todos los elementos del conjunto que invoca que son mayores a *lowerBound*. Si *i* es **true**, entonces un elemento igual a **lowerBound** sería incluido en el resultado. El conjunto resultante es una referencia al conjunto que invoca.

TABLA 17-4 Los métodos definidos por **NavigableSet** *(continuación)*

Una excepción de tipo **ClassCastException** se genera cuando un objeto no es compatible con los elementos en el conjunto. Una excepción de tipo **NullPointerException** es generada si se intenta utilizar un objeto null y null no está permitido en el conjunto. Una excepción de tipo **IllegalArgumentException** se genera si un argumento inválido es utilizado.

La interfaz Queue

La interfaz **Queue** extiende de la interfaz **Collection** y declara el comportamiento de una fila, la cual es a menudo una lista que implementa un comportamiento de primero en entrar – primero en salir. Sin embargo, existen tipos de filas donde el orden es un criterio adicional a considerar al momento de insertar y borrar elementos. **Queue** es una interfaz genérica declarada como;

interface Queue<E>

Método	Descripción
E element( )	Devuelve el elemento al inicio de la fila. El elemento no es removido de la fila. Se genera una excepción de tipo **NoSuchElementException** si la fila está vacía.
Boolean offer(E *obj*)	Intenta añadir *obj* a la fila. Devuelve **true** si *obj* fue añadido con éxito a la fila y **false** en caso contrario.
E peek( )	Devuelve el elemento al inicio de la fila. Devuelve **null** si la fila está vacía. El elemento no es removido.
E remove( )	Remueve el elemento al inicio de la fila y lo devuelve. Genera una excepción de tipo **NoSuchElementException** si la fila está vacía.

TABLA 17-5 Métodos definidos por **Queue**

Donde, **E** especifica el tipo de objetos que la fila contendrá. Los métodos definidos por Queue se muestran en la Tabla 17-5.

Varios métodos generan una excepción de tipo **ClassCastException** cuando un objeto no es compatible con los elementos de la fila. Una excepción **NullPointerException** se genera

si se intenta almacenar un objeto **null** y los elementos **null** no están permitidos en la fila. Una excepción de tipo **IllegalArgumentException** se genera si un argumento no válido es proporcionado a un método. Una excepción **IllegalStateException** se genera si se intenta añadir un elemento a una fila de tamaño fijo y ésta se encuentra llena. Una excepción de tipo **NoSuchElementException** se genera si se intenta quitar un elemento de una fila vacía.

A pesar de su simplicidad, **Queue** ofrece varios aspectos interesantes. Primero, los elementos sólo pueden ser removidos desde el inicio de la fila. Segundo, existen dos métodos que obtienen y remueven elementos: **poll( )** y **remove( ).** La diferencia entre ellos es que **poll( )** devuelve **null** si la fila está vacía mientras que **remove( )** lanza una excepción. Tercero, existen dos métodos, **element( )** y **peek( )**, que obtienen pero no remueven el elemento al inicio de la fila. Difieren únicamente en que **element( )** genera una excepción si la fila está vacía, pero **peek( )** devuelve **null**. Finalmente, observe que **offer( )** sólo intenta añadir un elemento a la fila. Debido a que algunas filas tienen tamaño fijo y podrían estar llenas, **offer( )** puede fallar.

La interfaz Dequeue

La interfaz **Dequeue** fue añadida por Java SE 6. Esta interfaz extiende de **Queue** y declara el comportamiento de una fila con doble final. Una fila con doble final puede trabajar como una fila estándar, primero en entrar – primero en salir, o como una pila último en entrar – primero en salir. **Dequeue** es una interfaz genérica que está declarada como:

```
interface Dequeue<E>
```

Donde, **E** especifica el tipo de objetos que la fila doble contendrá. Adicionalmente a los métodos que se heredan de **Queue**, la interfaz **Dequeue** añade los métodos listados en la Tabla 17-6. Varios métodos generan una excepción **ClassCastException** cuando un objeto es incompatible con los elementos en la fila doble. Una excepción de tipo **NullPointerException** se genera si se intenta almacenar un objeto **null** y los elementos **null** no están permitidos en la fila doble. Una excepción **IllegalArgumentException** se genera si un argumento no válido es proporcionado a un método. Una excepción **IllegalStateException** se genera si se intenta añadir un elemento a una fila doble de tamaño fijo y ésta se encuentra llena. Una excepción de tipo **NoSuchElementException** se genera si se intenta quitar un elemento de una fila doble vacía.

Nótese que **Dequeue** incluye los métodos **push( )** y **pop( )**. Estos métodos permiten a **Dequeue** funcionar como una pila. Además observe el método **descendingIterator( )**, el cual devuelve un iterador que devuelve elementos en orden inverso. En otras palabras, devuelve un iterador que se mueve del final al inicio de la colección. La implementación de un **Dequeue** puede realizarse *limitando su tamaño* a un número predefinido de elementos.

Cuando esto se hace y la inserción de un elemento nuevo falla, es posible manejar la falla de dos formas. Primero, métodos como **addFirst( )** y **addLast( )** generan una excepción de tipo **IllegalStateException** si una fila doble de tamaño predefinido está llena.

Método	Descripción
void addFirst(E *obj*)	Añade *obj* al inicio de la fila doble. Genera una excepción **IllegalStateException** si la fila tiene tamaño predefinido y no existe espacio disponible.
void addLast(E *obj*)	Añade *obj* al final de la fila doble. Genera una excepción **IllegalStateException** si la fila tiene tamaño predefinido y no existe espacio disponible.

TABLA 17-6 Métodos definidos por **Dequeue**

Método	Descripción
Iterator<E> descendingIterator( )	Devuelve un iterador que se mueve desde el final al inicio de la fila doble. En otras palabras, devuelve un iterador inverso.
E getFirst( )	Devuelve el primer elemento de la fila doble. El objeto no es removido de la fila. Este método genera una excepción **NoSuchElementException** si la fila doble está vacía.
E getLast( )	Devuelve el último elemento en la fila doble. El objeto no es removido de la fila doble. Genera una excepción de tipo **NoSuchElementException** si la fila doble está vacía.
boolean offerFirst(E *obj*)	Intenta añadir *obj* al inicio de la fila doble. Devuelve **true** si *obj* fue añadido y **false** en caso contrario. Este método devuelve **false** cuando se intenta añadir *obj* a una fila doble de tamaño predefinido que está llena.
boolean offerLast(E *obj*)	Intenta añadir *obj* al final de la fila doble. Devuelve **true** si *obj* fue añadido y **false** en caso contrario.
E peekFirst( )	Devuelve el elemento al inicio de la fila doble. Devuelve **null** si la fila doble está vacía. El objeto devuelto no es removido.
E peekLast( )	Devuelve el elemento al final de la fila doble. Devuelve **null** si la fila doble está vacía. El objeto devuelto no es removido.
E pollFirst( )	Devuelve y remueve el elemento al inicio de la fila doble. Devuelve **null** si la fila doble está vacía.
E pollLast( )	Devuelve y remueve el elemento al final de la fila doble. Devuelve **null** si la fila doble está vacía.
E pop( )	Devuelve y remueve el elemento al inicio de la fila doble. Genera una excepción **NoSuchElementException** si la fila doble está vacía.
void push(E *obj*)	Añade *obj* al inicio de la fila doble. Genera una excepción **IllegalStateException** si la fila tiene tamaño predefinido y no existe espacio disponible.
E removeFirst( )	Devuelve y remueve el elemento al inicio de la fila doble. Genera una excepción de tipo **NoSuchElementException** si la fila doble está vacía.
boolean removeFirstOcurrence (Object *obj*)	Remueve la primera ocurrencia de *obj* de la fila doble. Devuelve **true** si el elemento es removido con éxito y **false** si la fila doble no contiene a *obj*.
E removeLast( )	Devuelve y elimina el elemento al final de la fila doble. Genera una excepción de tipo **NoSuchElementException** si la fila doble está vacía.
boolean removeLastOcurrence (Object *obj*)	Devuelve la última ocurrencia de *obj* de la fila doble. Devuelve **true** si el elemento es removido con éxito y **false** si la fila doble no contiene a *obj*.

TABLA 17-6 Métodos definidos por **Dequeue** *(continuación)*

Segundo, los métodos como **offerFirst( )** y **offerLast( )** devuelven false si el elemento no puede ser añadido.

Las clases de la estructura de colecciones

Ahora que nos hemos familiarizado con las interfaces de colección, estamos listos para examinar las clases estándar que las implementan. Algunas de las clases proporcionan implementaciones

completas que se pueden usar tal cual. Otras son abstractas y proporcionan el esqueleto de implementaciones que se usan como puntos de partida para crear colecciones concretas. Ninguna de las clases de colección está sincronizada, pero, como veremos más adelante en este capítulo, es posible obtener versiones sincronizadas.

Las clases de colección estándar se resumen en la siguiente tabla:

Clase	Descripción
AbstractCollection	Implementa la mayor parte de la interfaz **Collection**.
AbstractList	Extiende **AbstractCollection** e implementa la mayor parte de la interfaz **List**.
AbstractQueue	Extiende **AbstractCollection** e implementa la mayor parte de la interfaz **Queue**.
AbstractSequentialList	Extiende **AbstractList** para su uso por una colección que utilice acceso secuencial a sus elementos, en vez de aleatorio.
LinkedList	Implementa una lista enlazada extendiendo **AbstractSequentialList**.
ArrayList	Implementa un arreglo dinámico extendiendo **AbstractList**.
ArrayDequeue	Implementa una fila dinámica con doble final extendiendo **AbstractCollection** e implementando la interfaz **Dequeue**. Añadido en Java SE 6.
AbstractSet	Extiende **AbstractCollection** e implementa la mayor parte de la interfaz **Set**.
EnumSet	Extiende **AbstractSet** para utilizarlo con elementos de tipo **enum**.
HashSet	Extiende **AbstracSet** para su uso con una tabla de dispersión.
LinkedHashSet	Extiende **HashSet** para permitir recorridos en orden de inserción.
PriorityQueue	Extiende **AbstractQueue** para soportar una fila basada en prioridades.
TreeSet	Implementa un conjunto almacenado en un árbol. Extiende **AbstractSet**.

Las siguientes secciones examinan las clases de colección concretas e ilustran su uso.

NOTA *Además de las clases de colección, se han rediseñado varias clases preexistentes como **Vector**, **Stack** y **Hashtable** para que soporten colecciones. Estas se examinarán más adelante en este capítulo.*

La clase ArrayList

La clase **ArrayList** extiende **AbstractList** e implementa la interfaz **List**. **ArrayList** es una clase genérica declarada como:

```
class ArrayList<E>
```

Donde, **E** especifica el tipo de objetos que la lista podría almacenar.

ArrayList soporta arreglos dinámicos que pueden crecer según se necesite. En Java, los arreglos estándar son de longitud fija. Después de creado un arreglo, su tamaño no puede aumentar ni disminuir, lo que significa que hay que saber de antemano cuántos elementos habrá en él. Pero a veces no se sabe exactamente que tan grande se necesita que sea un arreglo hasta el tiempo de ejecución. Para resolver esta situación, la estructura de colecciones define **ArrayList**. En esencia, una **ArrayList** es un arreglo de longitud variable que almacena referencias a objeto. Esto es, un **ArrayList** puede aumentar o disminuir de tamaño dinámicamente.

Los **ArrayList** se crean con un tamaño inicial. Cuando este tamaño se excede, la colección se agranda automáticamente. Cuando se quitan objetos, el arreglo puede disminuir de tamaño.

NOTA *Los arreglos dinámicos son soportados también por la clase* ***Vector****, la cual se describe más adelante en este capítulo.*

ArrayList tiene los constructores siguientes:

ArrayList( )
ArrayList(Collection<? extends E> *c*)
ArrayList(int *capacidad*)

El primer constructor construye un ArrayList vacío. El segundo construye un ArrayList que se inicializa con los elementos de la colección *c*. El tercero construye un ArrayList que tiene la *capacidad* inicial especificada. La capacidad es el tamaño del arreglo subyacente que se utiliza para almacenar los elementos. La capacidad crece automáticamente según se añaden elementos a la lista.

El siguiente programa muestra un uso sencillo de **ArrayList**. Se crea un **ArrayList**, y luego se le añaden objetos de tipo **String**. Recuérdese que una cadena entrecomillada se traduce a un objeto **String**. Finalmente la lista se muestra, algunos de los elementos se remueven y la lista se muestra de nuevo.

```
// Ejemplo con ArrayList.
import java.util.*;

class ArrayListDemo {
  public static void main(String args[]) {
    // crea un ArrayList
    ArrayList al = new ArrayList<String>();

    System.out.println("Tamaño inicial de al: " +
                       al. size());

    //  añadir elementos al ArrayList
    al.add("C");
    al.add("A");
    al.add("E");
    al.add("B");
    al.add("D");
    al.add("F");
    al.add(l, "A2");

    System.out.println("Tamaño de a1 después de las adiciones: " +
                       al.size());

    // mostrar el ArrayList
    System.out.println{"Contenido de al: " + al);

    // quitar elementos del ArrayList
    al.remove("F");
    al.remove(2);

    System.out.println("Tamaño de a1 después de quitar elementos: " +
                       al.size());

    System.out.println("Contenido de al: " + al);
  }
}
```

La salida de este programa se muestra aquí:

```
Tamaño inicial de a1: 0
Tamaño de a1 después de las adiciones: 7
Contenidos de a1: [C, A2, A, E, B, D, F]
Tamaño de a1 después de quitar elementos: 5
Contenidos de a1: [C, A2, E, B, D]
```

Nótese que el objeto a1 comienza vacía y va creciendo según se le añaden elementos. Cuando se quitan elementos, su tamaño se reduce.

En el ejemplo precedente, los contenidos de una colección se muestran utilizando la conversión por omisión proporcionada por el método **toString( )**, heredado de **AbstractCollection**. Aunque esto basta para programas y ejemplos cortos, rara vez se utiliza este método para mostrar los contenidos de una colección en el mundo real. Habitualmente el programador proporciona sus propias rutinas de salida. Sin embargo, aún usaremos la salida por omisión producida por **toString( )** para algunos de los siguientes ejemplos.

Aunque la capacidad de un objeto **ArrayList** aumenta automáticamente según se almacenan objetos en él, se puede aumentar su capacidad manualmente llamando al método **ensureCapacity( )**. Esto puede ser deseable si se conoce de antemano que más adelante se almacenarán en la colección muchos más elementos de los que actualmente puede contener. Aumentando su capacidad una vez, al inicio, se pueden evitar distintas reasignaciones de memoria más adelante. Dado que las reasignaciones son costosas en términos de tiempo, evitar reasignaciones innecesarias mejora el rendimiento. La firma de **ensureCapacity( )** se muestra a continuación:

void ensureCapacity(int *cap*)

Donde *cap* es la nueva capacidad.

A la inversa, para reducir el tamaño del arreglo subyacente a un objeto **ArrayList** para que su tamaño sea exactamente igual al número de elementos que actualmente contiene, se llama al método **trimToSize( )** de la siguiente forma:

void trimToSize( )

Obtención de un arreglo a partir de un ArrayList

Cuando se trabaja con **ArrayList**, en ocasiones se desea obtener un arreglo con los contenidos de la lista. Como ya se ha explicado, esto se puede lograr llamando al método **toArray( )** definido en **Collection**. Existen diferentes razones por las cuales se puede desear convertir una colección en un arreglo, como:

- Conseguir tiempos más rápidos de procesamiento para ciertas operaciones.
- Pasar un arreglo a un método que no está sobrecargado para aceptar una colección.
- Integrar el código de usuario más nuevo, basado en las colecciones, con código preexistente que no entiende de colecciones.

Cualquiera que sea la razón, convertir una **ArrayList** en un arreglo es algo trivial.

Como se explicó antes, se tienen dos versiones del método **toArray( )**, los cuales se muestran aquí nuevamente por comodidad.

Object[ ] toArray( )
<T> T[ ] toArray(T *arreglo*[ ])

El primer método devuelve un arreglo de elementos tipo **Object**. El segundo devuelve un arreglo de elementos que tienen el tipo definido en **T**. Normalmente, la segunda forma es más cómoda porque devuelve un arreglo de un tipo adecuado. El siguiente programa demuestra el uso del método toArray( ).

```
// Convertir una ArrayList en un arreglo.
import java.util.*;

class ArrayListToArreglo{
  public static void main{String args[]) {
    // Crear un ArrayList
    ArrayList al = new ArrayList();

    // Añadir elementos
    al.add(1);
    al.add(2);
    al.add(3);
    al.add(4);

    System.out.println("Contenido de al: " + al);

    // Obtener el arreglo
    Integer ia[] = new Integer [al.size()];
    ia = a1.toArray(ia);

    int sum = 0;

    // Sumar el arreglo
    for(int i : ia) sum += i;

    System.out.println("La suma es: " + sum);
  }
}
```

La salida del programa se muestra aquí:

```
Contenidos de al: [1, 2, 3, 4]
La suma es: 10
```

El programa comienza creando una colección de enteros. A continuación, se llama a **toArray( )**, que obtiene un arreglo de objetos tipo **Integer**. Luego, los contenidos de este arreglo se suman utilizando un ciclo estilo **for-each**.

Otro elemento interesante del programa anterior es el siguiente. Como sabemos, las colecciones sólo pueden almacenar referencias, no valores, a tipos primitivos. Sin embargo, autoboxing hace posible pasar valores de tipo **int** al método **add( )** sin tener que envolverlo manualmente en un objeto de tipo **Integer**. Autoboxing realiza una envoltura automática. En este sentido, autoboxing mejora significativamente la facilidad con la cual las colecciones pueden ser utilizadas para almacenar valores primitivos.

La clase LinkedList

La clase **LinkedList** extiende de **AbstractSequentialList** e implementa la interfaz **List**, **Dequeue** y **Queue**. **LinkedList** proporciona una estructura de datos de tipo lista enlazada. **LinkedList** es una clase genérica declarada como:

```
class LinkedList<E>
```

Donde, **E** especifica el tipo de los objetos que la lista almacenará. **LinkedList** tiene los siguientes dos constructores:

LinkedList( )
LinkedList(Collection<? extends E> c)

El primer constructor crea una lista enlazada vacía. El segundo construye una lista enlazada que se inicializa con los elementos de la colección *c*.

Debido a que **LinkedList** implementa la interfaz **Dequeue** tenemos acceso a los métodos definidos por **Dequeue**. Por ejemplo, para añadir elementos al inicio de una lista se puede usar **addFirst( )** u **offerFirst( )**. Para añadir elementos al final de la lista se puede usar **addLast( )** u **offerLast( )**. Para obtener el primer elemento, se puede usar **getFirst( )** o **peekFirst( )**. Para obtener el último elemento se utiliza **getLast( )** o **peekLast( )**. Para remover el primer elemento, se utiliza **removeFirst( )** o **pollFirst( )**. Para eliminar el último elemento se utiliza **removeLast( )** o **pollLast( )**.

El siguiente programa ilustra varios de los métodos soportados por **LinkedList**:

```
// Ejemplo con LinkedList.
import java.util.*;

class LinkedListDemo {
  public static void main(String args[]) {
    // crear una lista enlazada
    LinkedList l1 = new LinkedList<String>();

    // añadir elementos a la lista vinculada
    l1.add("F");
    l1.add("B");
    l1.add("D");
    l1.add("E");
    l1.add("C");
    l1.addLast("Z");
    l1.addFirst("A");

    l1.add(1, "A2");

    System.out.println("Contenido original de l1: " + l1);

    // quitar elementos de la lista enlazada
    l1.remove("F");
    l1.remove(2);

    System.out.println("Contenido de l1 tras quitar elementos:
                       + l1);

    // quitar los elementos primero y último

    l1.removeFirst();
    l1.removeLast();

    System.out.println("l1 tras quitar el primero y el último:
                       + l1);

    // obtener y asignar un valor
    String val = l1.get(2);
    l1.set(2, val + " cambiado");

    System.out.println("l1 tras el cambio: " + l1);
  }
}
```

La salida de este programa es ésta:

```
Contenido original de l1: [A, A2, F, B, D, E, C, Z]
Contenido de l1 tras quitar elementos: [A, A2, D, E, C, Z]
l1 tras quitar el primero y el último: [A2, D, E, C]
l1 tras el cambio: [A2, D, E cambiado, C]
```

Como **LinkedList** implementa la interfaz **List**, las llamadas a **add(E)** añaden elementos al final de la lista, al igual que **addLast( )**. Para insertar elementos en una posición específica, ha de usarse la forma **add(int,E)** de **add( )**, como se ilustra con la llamada a **add(l,"A2")** en el ejemplo.

Nótese cómo el tercer elemento de **ll** se cambia empleando llamadas a **get( )** y **set( )**. Para obtener el valor actualizado de un elemento, ha de pasarse a **get( )** el índice en que se almacena dicho elemento. Para asignar un valor nuevo a ese índice, ha de pasarse a **set( )** el índice y su nuevo valor.

La clase HashSet

HashSet extiende **AbstractSet** e implementa la interfaz **Set**. Crea una colección que usa una tabla de dispersión para el almacenamiento. **HashSet** es una clase genérica que está declarada como:

```
class HashTable<E>
```

Donde, **E** especifica el tipo de objetos que la colección almacenará.

Como muchos lectores sabrán, una tabla de dispersión almacena información usando un mecanismo llamado dispersión. Con el uso de la *dispersión*, la información de una clave se utiliza para determinar un valor único, llamado su *código de dispersión*. El código de dispersión se utiliza entonces como el índice donde se almacenan los datos asociados con la clave. La transformación de la clave en su código de dispersión se lleva a cabo automáticamente —nunca se ve el código de dispersión—. Además, el código de dispersión no puede indexar directamente la tabla de dispersión. La ventaja de la dispersión es que permite mantener constante el tiempo de ejecución de operaciones básicas, como **add( )**, **contains( )**, **remove( )** y **size( )**, incluso para conjuntos grandes.

HashSet define los siguientes constructores:

```
HashSet( )
HashSet(Collection <? extends c)
HashSet(int capacidad)
HashSet(int capacidad, float llenar)
```

La primera forma construye un conjunto de dispersión por omisión. La segunda forma lo inicializa usando los elementos de *c*. La tercera inicializa la capacidad del conjunto de dispersión a *capacidad*. La cuarta forma inicializa a partir de sus argumentos tanto la capacidad como la razón de llenado (también llamada *capacidad de carga*) del conjunto de dispersión. La razón de llenado debe estar comprendida entre 0.0 y 1.0, y determina qué tan de lleno puede estar el conjunto de dispersión antes de que se aumente de tamaño. Específicamente, cuando el número de elementos es mayor que la capacidad del conjunto de dispersión multiplicada por su razón de llenado, el conjunto de dispersión se expande. Para constructores que no toman razón de llenado, se utiliza 0.75.

HashSet no define ningún método además de los ya proporcionados por sus superclases e interfaces.

Es importante mencionar que un **conjunto de dispersión** no garantiza el orden de sus elementos, porque el proceso de dispersión no se suele prestar a la creación de conjuntos

ordenados. Si hace falta almacenamiento ordenado, entonces es mejor elegir otra colección, como por ejemplo **TreeSet**.

Aquí hay un ejemplo que ilustra **HashSet**:

```
// Ejemplo con HashSet.
import java.util.*;

class HashSetDemo {
  public static void main(String args[]) {
    // crear un conjunto de dispersión
    HashSet<String> hs = new HashSet<String>();

    // añadir elementos al conjunto de dispersión
    hs.add("B");
    hs.add("A");
    hs.add("D");
    hs.add("E");
    hs.add("C");
    hs.add("F");

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

Ésta es la salida del programa:

```
[D, A, F, C, B, E]
```

Como se ha explicado, los elementos no se almacenan en orden y la salida exacta puede variar.

La clase LinkedHashSet

La clase **LinkedHashSet** extiende **HashSet** y no añade miembros propios. Es una clase genérica declarada como:

class LinkedHashSet<E>

Aquí, **E** especifica el tipo de objetos que el conjunto almacenará. Sus constructores son iguales a los de **HashSet**.

LinkedHashSet mantiene una lista enlazada de los elementos que se agregan al conjunto, en el orden en que son agregados. Esto permite recorrer el conjunto en el orden en que los elementos fueron agregados. Esto es, cuando se recorre el **LinkedHashSet** utilizando un iterador, los elementos serán devueltos en el orden en el cual fueron insertados. Éste es también el orden en que serán contenidos en la cadena devuelta por **toString( )** cuando es llamado con un objeto **LinkedHashSet**. Para ver el efecto de un **LinkedHashSet**, intente sustituir **LinkedHashSet** por **HashSet** en el programa anterior. La salida será:

```
[B, A, D, E, C, F]
```

Éste es el orden en el cual los elementos fueron insertados.

La Clase TreeSet

TreeSet extiende de **AbstractSet** e implementa la interfaz **NavigableSet**. **TreeSet** crea una colección que utiliza un árbol para el almacenamiento de datos. Los objetos se almacenan ordenados, en orden ascendente. Los tiempos de acceso y recuperación son bastante rápidos,

lo que hace de **TreeSet** una excelente elección cuando se almacenan grandes cantidades de información ordenada que debe encontrarse rápidamente.

TreeSet es una clase genérica que se declara como:

Class TreeSet<E>

Donde, **E** especifica el tipo de objetos que el conjunto almacenará.

TreeSet define los siguientes constructores:

TreeSet( )
TreeSet(Collection<? extends E> *c*)
TreeSet(Comparator <? super E> *comp*)
TreeSet(SortedSet<E> *ss*)

La primera forma construye un **TreeSet** vacío que se ordenará en orden ascendente de acuerdo con el orden natural de sus elementos. La segunda forma construye un **TreeSet** que contiene los elementos de *c*. La tercera forma construye un **TreeSet** vacío que se ordenará de acuerdo con el comparador especificado por *comp*. (Los comparadores se describen más adelante en este capítulo.) La cuarta forma construye un **TreeSet** que contiene los elementos de *ss*.

Aquí hay un ejemplo que muestra el uso de un **TreeSet**:

```
// Ejemplo con TreeSet.
import java.util.*;

class TreeSetDemo {
  public static void main(String args[]) {
    // Crear un conjunto en árbol
    TreeSet<String> ts = new TreeSet<String>();

    // Añadir elementos al árbol
    ts.add("C");
    ts.add("A");
    ts.add("B");
    ts.add("E");
    ts.add("F");
    ts.add("D");

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

La salida del programa se muestra a continuación:

```
[A, B, C, D, E, F]
```

Como hemos explicado, dado que **TreeSet** almacena sus elementos en un árbol, se ordenan automáticamente.

Debido a que **TreeSet** implementa la interfaz **NavigableSet** (la cual fue añadida por Java SE 6), se pueden utilizar los métodos definidos por **NavigableSet** para recuperar elementos de un **TreeSet**. Por ejemplo, considerando el programa anterior, la siguiente sentencia utiliza al método **subset( )** para obtener un subconjunto de **ts** que contiene el elemento entre **C** (incluyéndola) y **F** (sin incluirla). Luego se despliega el conjunto resultante.

```
System.out.println(ts.subset( "C", "F"));
```

La salida producida por la línea anterior es:

```
[C, D, E]
```

De forma similar se puede experimentar con otros métodos definidos por **NavigableSet**.

La clase PriorityQueue

PriorityQueue extiende a **AbstractQueue** e implementa la interfaz **Queue**. **PriorityQueue** crea una fila basada en prioridades acorde con un comparador definido. **PriorityQueue** es una clase genérica declarada como:

```
class PriorityQueue<E>
```

Donde, **E** especifica el tipo de objetos almacenados en la fila. **PriorityQueue** crece de manera dinámica según sea necesario.

PriorityQueue define seis constructores:

```
PriorityQueue( )
PriorityQueue(int capacidad)
PriorityQueue(int capacidad, Comparator<? super E> comp)
PriorityQueue(Collection<? extends E>c)
PriorityQueue(PriorityQueue<? extends E>c)
PriorityQueue(SortedSet<? extends E>c)
```

El primer constructor construye una fila vacía. Su capacidad inicial es 11. El segundo constructor construye una fila que tiene la capacidad inicial especificada en el parámetro capacidad. El tercer constructor construye una fila con la capacidad y el comparador especificados. Los últimos tres constructores crean filas que se inicializan con los elementos de la colección *c*, pasada como argumento. En todos los casos, la capacidad crece automáticamente conforme se agregan elementos.

Si ningún comparador se especifica cuando se construye una **PriorityQueue**, se utiliza el comparador por omisión para el tipo de dato almacenado en la fila. El comparador por omisión ordenara la fila en orden ascendente. Así, el inicio de la fila contendrá al menor de los valores. Sin embargo, proporcionando un comparador personalizado, se puede especificar un esquema diferente de ordenamiento. Por ejemplo, cuando se almacenan elementos que incluyen horas o fechas, se puede priorizar la fila para que el elemento más antiguo sea el primero de la fila.

Se puede obtener una referencia al comparador utilizado por una **PriorityQueue** llamando a su método **comparator( )**, definido como:

```
Comparator<? super E> comparator( )
```

Este método devuelve al comparador. Si la fila que invoca utiliza ordenamiento natural, el método devuelve **null**.

Es importante considerar que aunque se puede recorrer una **PriorityQueue** utilizando un iterador, el orden de dicha iteración no está definido. Para utilizar adecuadamente una **PriorityQueue** se deben llamar a los métodos **offer( )** y **poll( )** definidos en la interfaz **Queue**.

La clase ArrayDequeue

Java SE 6 añade la clase **ArrayDequeue**, la cual extiende de **AbstractCollection** e implementa la interfaz **Dequeue**. Esta clase no añade métodos propios. **ArrayDequeue** crea un arreglo dinámico sin restricciones de capacidad. La interfaz **Dequeue** soporta implementaciones que restringen la capacidad de almacenamiento, sin embargo establecer dicha restricción es opcional. **ArrayDequeue** es una clase genérica declarada como:

```
class ArrayDequeue<E>
```

Aquí, **E** especifica el tipo de objetos almacenados en la colección.

ArrayDequeue define los siguientes constructores:

ArrayDequeue( )

ArrayDequeue(int *tamaño*)

ArrayDequeue(Collection<? extends E> *c*)

El primer constructor construye una fila doble vacía. Su capacidad inicial es 16. El segundo constructor construye una fila doble que cuenta con la capacidad inicial definida por el argumento tamaño. El tercer constructor crea una fila doble que está inicializada con los elementos de la colección dada en el argumento *c*. En todos los casos, la capacidad crece conforme se necesita para manejar los elementos añadidos a la fila doble.

El siguiente programa demuestra el uso de **ArrayDequeue** utilizando esta colección para imitar el comportamiento de una pila:

```
// Ejemplo con ArrayDequeue.
import java.util.*;

class ArrayDequeueDemo {
  public static void main(String args[]) {
    // Crear un ArrayDequeue
    ArrayDequeue<String> adq = new ArrayDequeue<String>();

    // Usar el ArrayDequeue como pila
    adq.push("A");
    adq.push("B");
    adq.push("D");
    adq.push("E");
    adq.push("F");

    System.out.print("Sacando elementos de la pila: ");

    while(adq.peek() != null)
      System.out.println(adq.pop() + " ");

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

La salida del programa se muestra a continuación:

```
Sacando elementos de la pila: F E D B A
```

La clase EnumSet

EnumSet extiende **AbstractSet** e implementa **Set**. Está hecho específicamente para ser usado con llaves de un tipo **enumerado**. Es una clase genérica declarada como:

class EnumSet<E extends Enum<E>>

Donde, **E** especifica a los elementos. Note que **E** debe extender **Enum<E>**, lo cual refuerza los requerimientos de que los elementos deben ser del tipo **enum** especificado.

EnumSet no define constructores. En lugar de ello, utiliza los métodos de fábrica mostrados en la Tabla 17-7 para crear objetos. Todos los métodos generan **NullPointerException** en caso de que se presente un problema. Los métodos **copyOf( )** y **range( )** pueden además generar una excepción de tipo **IllegalArgumentException**. Observe que el método **of( )** está sobrecargado

varias veces en busca de una mayor eficiencia. Pasar un número conocido de argumentos genera una aplicación más veloz que aquella que utiliza parámetros vararg cuando el número de argumentos es pequeño.

Acceso a una colección por medio de un iterator

A menudo se desea ejecutar un recorrido a lo largo de los elementos de una colección. Por ejemplo, se puede querer mostrar cada elemento. Una forma de hacer esto es empleando un *iterador*, un objeto que implementa la interfaz **Iterator** o la interfaz **ListIterator**. Un **Iterator** proporciona un ciclo a lo largo de una colección, obteniendo o quitando elementos. **ListIterator**

Método	Descripción
static <E extends Enum<E>> EnumSet<E> allOf(Class<E> *t*)	Crea un **EnumSet** que contiene los elementos en la enumeración especificada por *t*.
static <E extends Enum<E>> EnumSet<E> complementOf(EnumSet<E> *e*)	Crea un **EnumSet** que incluye los elementos que no están almacenados en *e*.
static <E extends Enum<E>> EnumSet<E> copyOf(EnumSet<E> *c*)	Crea un **EnumSet** a partir de los elementos almacenados en *c*.
static <E extends Enum<E>> EnumSet<E> copyOf(Collection<E> *c*)	Crea un **EnumSet** a partir de los elementos almacenados en *c*.
static <E extends Enum<E>> EnumSet<E> noneOf(Class<E> *t*)	Crea un **EnumSet** que contiene los elementos que no estén en la enumeración especificada por *t*, el cual está vacío por definición.
static <E extends Enum<E>> EnumSet<E> of(E v, E … *vargs*)	Crea un **EnumSet** que contiene *v* y cero o más valores adicionales de enumeración.
static <E extends Enum<E>> EnumSet<E> of(E *v*)	Crea un **EnumSet** que contiene *v*.
static <E extends Enum<E>> EnumSet<E> of(E *v1*, E *v2*)	Crea un **EnumSet** que contiene *v1* y *v2*.
static <E extends Enum<E>> EnumSet<E> of(E *v1*, E *v2*, E *v3*)	Crea un **EnumSet** que contiene de *v1* a *v3*.
static <E extends Enum<E>> EnumSet<E> of(E *v1*, E *v2*, E *v3*, E *v4*)	Crea un **EnumSet** que contiene de *v1* a *v4*.
static <E extends Enum<E>> EnumSet<E> of(E *v1*, E *v2*, E *v3*, E *v4*, E *v5*)	Crea un **EnumSet** que contiene de *v1* a *v5*.
static <E extends Enum<E>> EnumSet<E> range(E *inicio*, E *fin*)	Crea un **EnumSet** que contiene los elementos en el rango especificado por *inicio* y *fin*.

TABLA 17-7 Los métodos definidos por **EnumSet**

Método	Descripción
Boolean hasNext( )	Devuelve **true** si existen más elementos. En otro caso devuelve **false**.
E next( )	Devuelve el siguiente elemento. Genera una excepción **NoSuchElementException** si no hay un siguiente elemento.
void remove( )	Remueve el elemento actual. Genera una excepción de tipo **IllegalStateException** si se intenta llamar a **remove( )** sin haber llamado antes a **next( ).**

TABLA 17-8 Los métodos definidos por **Iterator**

extiende **Iterator** para permitir atravesar una lista bidireccionalmente, así como la modificación de elementos. **Iterator** y **ListIterator** son interfaces genéricas declaradas como:

```
interface Iterator<E>
interface ListIterator<E>
```

Donde, **E** especifica el tipo de objetos que serán iterados. La interfaz **Iterator** declara los métodos mostrados en la tabla 17-8. Los métodos declarados por **ListIterator** se muestran en la Tabla 17-9. En ambos casos, las operaciones que modifican la colección subyacente son opcionales. Por ejemplo, **remove( )** generará una excepción de tipo **UnsupportedOperationException** cuando se utiliza con una colección de sólo lectura.

Uso de un iterador

Antes de poder acceder a una colección por medio de un iterador, se debe obtener uno. Cada una de las clases de colecciones proporciona un método **iterator( )** que devuelve un iterador al inicio de la colección. Utilizando este objeto iterador, se puede acceder a cada elemento de la colección, uno a uno.

Método	Descripción
void add(E *obj*)	Inserta *obj* en la lista delante del elemento devuelto por la siguiente llamada a **next( )**.
boolean hasNext( )	Devuelve **true** si existe un elemento siguiente. De lo contrario, devuelve **false**.
boolean hasPrevious( )	Devuelve **true** si existe un elemento anterior. De lo contrario, devuelve **false**.
E next( )	Devuelve el siguiente elemento. Si no existe un elemento siguiente, se produce una excepción de tipo **NoSuchElementException**.
int nextIndex( )	Devuelve el índice del siguiente elemento. Si no existe siguiente elemento, devuelve el tamaño de la lista.
Object previous( )	Devuelve el elemento anterior. Si no existe elemento anterior, se produce una excepción de tipo **NoSuchElementException**.
int previousIndex( )	Devuelve el índice del elemento anterior. Si no hay elemento anterior, devuelve –1.
void remove( )	Quita el elemento actual de la lista. Se produce una excepción **IllegalStateException** si se llama a **remove( )** antes de llamar a **next( )** o **previous( )**.
void set(E *obj*)	Asigna *obj* al elemento actual. Este es el último elemento devuelto por una llamada a **next( )** o a **previous( )**.

TABLA 17-9 Los métodos declarados por **ListIterator**

En general, para utilizar un iterador para recorrer el contenido de una colección, se han de seguir estos pasos:

1. Obtener un iterador al principio de la colección llamando al método **iterator( )** de la colección.
2. Programar un ciclo que llame a **hasNext( ).** Hacer que el ciclo itere mientras **hasNext( )** devuelve **true**.
3. Dentro del ciclo, se obtiene cada elemento llamando al método **next( ).**

Para colecciones que implementan **List**, también se puede obtener un iterador llamando a **listIterator( )**. Como se ha explicado, un iterador de lista permite acceder a la colección en ambas direcciones, hacia delante o hacia atrás, así como modificar un elemento. Por lo demás,

ListIterator se utiliza justo como Iterator. Aquí hay un ejemplo que implementa estos pasos, ilustrando el uso de las interfaces **Iterator** y **ListIterator**. Se utilizan con un objeto **ArrayList**, pero los mismos principios generales se aplican a cualquier tipo de colección. Por supuesto, **ListIterator** está disponible sólo para aquellas colecciones que implementen la interfaz **List**.

```
// Ejemplo con iteradores.
import java.util.*;

class IteratorDemo {
  public static void main(String args[]) {
    // crear una lista de arreglos
    ArrayList<String> al = new ArrayList<String>();

    // añadir elementos al ArrayList
    al.add("C");
    al.add("A");
    al.add("E");
    al.add("B");
    al.add("D");
    al.add("F");

    // usar el iterador para mostrar los contenidos de al
    System.out.print("Contenidos originales de al: ");
    Iterator<String> itr = al.iterator();
    while(itr.hasNext()) {
      String element = itr.next();
      System.out.print(element + " ");
    }
    System.out.println();

    // modificar los objetos iterados
    ListIterator litr = al.listIterator();
    while(litr.hasNext()) {
      String element = litr.next();
      litr.set(element + "+");
    }

    System.out.print("Contenidos de al modificados: ");
    itr = al.iterator();
    while(itr.hasNext()) {
    String element = itr.next();
    System.out.print(element + " ");
    }
    System.out.println();

    // ahora, mostrar la lista en orden inverso
    System.out.print("Lista modificada en orden inverso: ");
    while(litr.hasPrevious()) {
      String element = litr.previous();
      System.out.print(element + " ");
    }
    System.out.println();
  }
}
```

La salida se muestra a continuación:

```
Contenidos originales de al: C A E B D F
```

```
Contenidos de al modificados: C+ A+ E+ B+ D+ F+
Lista modificada en orden inverso: F+ D+ B+ E+ A+ C+
```

Prestemos especial atención a cómo se muestra la lista en orden inverso. Después de que la lista es modificada, **litr** apunta al final de la lista. Recordemos que **litr.hasNext( )** devuelve **falso** cuando se ha alcanzado el final de la lista. Para recorrer la lista hacia atrás, el programa sigue utilizando **litr**, pero esta vez comprueba si existe un elemento precedente. Mientras exista, el elemento se obtiene y se muestra.

for-each como alternativa de los iteradores

Si no se desea modificar el contenido de una colección u obtener elementos en orden inverso, entonces la versión for-each del ciclo es una alternativa más cómoda que un iterador para recorrer la colección. Recuerde que un ciclo **for** puede recorrer cualquier colección de objetos que implemente la interfaz **Iterable**. Debido a que todas las clases de colección implementan esta interfaz, todas pueden ser recorridas utilizando un ciclo **for**.

El siguiente ejemplo utiliza un ciclo **for** para sumar el contenido de una colección:

```
// Ejemplo de recorrido de colecciones con ciclos for-each
import java.util.*;

class ForEachDemo {
  public static void main(String args[]) {
    // crear un ArrayList de enteros
    ArrayList<Integer> vals = new ArrayList<Integer>();

    // añadir elementos al ArrayList
    vals.add(1);
    vals.add(2);
    vals.add(3);
    vals.add(4);
    vals.add(5);

    // usar un ciclo for para mostrar los elementos
    System.out.print("Contenidos originales de vals: ");
    for (int v: vals)
      System.out.print(v + " ");

    System.out.println();

    // Ahora se suman los valores con otro ciclo
    int sum = 0;
    for (int v: vals)
      sum += v;

    System.out.println("La suma de los valores es: " + sum );
  }
}
```

La salida del programa es la siguiente:

```
Contenidos originales de vals: 1 2 3 4 5
La suma de los valores es: 15
```

Como podemos ver, el ciclo **for** es notablemente más corto y simple de usar que la estrategia basada en iteradores. Sin embargo, éste sólo puede ser utilizado para recorrer una colección hacia delante y no es posible modificar el contenido de una colección utilizando for-each.

Almacenamiento de clases definidas por el usuario en colecciones

Por simplicidad, los ejemplos anteriores almacenaban objetos definidos por clases propias de Java, como **String** o **Integer**, en colecciones. Por supuesto, las colecciones no se limitan al almacenamiento de objetos definidos con clases incorporadas en Java. Más bien al contrario. El poder de las colecciones es que pueden almacenar cualquier tipo de objeto, incluyendo objetos de clases creadas por el programador. Por ejemplo, considérese el siguiente ejemplo que utiliza una **LinkedList** para almacenar direcciones postales:

```
// Un ejemplo sencillo de lista de direcciones postales.
import java.util.*;

class Address {
  private String nombre;
  private String calle;
  private String ciudad;
  private String estado;
  private String codigo;

  Address(String n, String s, String c, String st, String cd) {
          nombre = n;
    calle = s;
    ciudad = c;
    estado = st;
    codigo = cd;
  }

  public String toString() {
    return nombre + "\n" + calle + "\n" +
           ciudad + " " + estado + " " + código;
  }
}

class MailList {
  public static void main(String args[]) {
    LinkedList<Address> m1 = new LinkedList<Address> ();

    // añadir elementos a la lista enlazada
    ml.add(new Address("José Luis Acosta" , "Cuevitas 110",
                       "México", "DF", "06110"));

    ml.add(new Address("Fabian Díaz", "Insurgentes 75",
                       "Monterrey", "NL", "61853"));

    ml.add(new Address("Leticia Hernández", "Av. Acueducto 1604",
                       "Zapopan", "JAL", "61820"));

    // desplegar la lista de direcciones
    For (Address elemento: m1)
      System.out.println(elemento + "\n");

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

La salida del programa es ésta:

```
José Luis Acosta
Cuevitas 110
México DF 06110
```

```
Fabian Díaz
Insurgentes 75
Monterrey NL 61853

Leticia Hernández
Av. Acueducto 1604
Zapopan JAL 61820
```

Aparte de almacenar una clase definida por el usuario en una colección, otra cosa importante sobre el programa anterior es que es bastante corto. Cuando se considera que establece una lista vinculada que puede almacenar, recuperar y procesar direcciones postales en unas 50 líneas de código, la potencia de la estructura de colecciones empieza a manifestarse. Como muchos lectores sabrán, si toda esta funcionalidad hubiera de programarse manualmente, el programa sería varias veces más grande.

La interfaz RandomAccess

La interfaz **RandomAccess** no contiene miembros. Sin embargo, al implementar esta interfaz una colección indica que soporta de forma eficiente el acceso aleatorio a sus elementos. Aunque una colección pueda soportar acceso aleatorio, ésta podría no hacerlo de forma eficiente. Revisando la interfaz **RandomAccess** el código puede determinar en tiempo de ejecución si una colección es adecuada para ciertas operaciones de acceso aleatorio, especialmente cuando se trata de colecciones grandes. Se puede emplear la palabra clave **instanceOf** para determinar si una clase implementa una interfaz. La interfaz **RandomAccess** es implementada por **ArrayList** y por la antigua clase **Vector**, entre otras.

Trabajo con mapas

Un *mapa* es un objeto que almacena asociaciones entre claves y valores, o *pares clave/valor*. Dada una clave, se puede encontrar su valor. Tanto las claves como los valores son objetos. La clave debe ser única, pero los valores pueden estar duplicados. Algunos mapas aceptan una clave **null** y valores **null**; otros no.

Un punto relevante acerca de los mapas que vale la pena mencionar es el hecho de que los mapas no implementan la interfaz **Iterable**. Esto significa que *no se puede* recorrer el mapa utilizando un ciclo **for-each**. Además para un mapa no es posible obtener un iterador. Sin embargo, como veremos pronto, es posible obtener una vista como colección de un mapa, la cual si permitiría el uso tanto de mapas como de ciclos **for**.

Las interfaces de Map

Debido a que las interfaces para trabajo con mapas definen el carácter y la naturaleza de los mapas, trataremos los mapas comenzando con sus interfaces. Las siguientes interfaces soportan mapas:

Interfaz	Descripción
Map	Mapea claves únicas a valores.
Map.Entry	Describe un elemento (un par clave/valor) en un mapa. Ésta es una clase interna de **Map**.
NavigableMap	Extiende **SortedMap** para manejar la recuperación de datos basado en búsquedas de proximidad (añadidas por Java SE 6).
SortedMap	Extiende **Map** para que las claves se mantengan en orden ascendente.

A continuación se analiza cada una de estas interfaces.

La interfaz Map

La interfaz **Map** mapea claves únicas a valores. Una *clave* es un objeto que se utilizará para recuperar un valor más adelante. Dadas una clave y un valor, se puede almacenar al valor en un objeto **Map**. Después de que se almacena el valor, se puede recuperar utilizando su clave. **Map** es una interfaz genérica y está definida como sigue:

```
interface Map<K,V>
```

Donde, **K** especifica el tipo de las claves y **V** especifica el tipo de los valores.

Los métodos declarados por **Map** se resumen en la Tabla 17-10. Varios métodos generan una excepción de tipo **ClassCastException** cuando un objeto es incompatible con los elementos en el mapa. Una excepción **NullPointerException** se genera si se intenta utilizar un objeto **null** y el uso de **null** no está permitido en el mapa. Una excepción de tipo **UnsupportedOperationException** se lanza cuando se intenta realizar cambios en un mapa no modificable. Una excepción de tipo **IllegalArgumentException** se lanza si se utiliza un argumento no válido.

Los mapas giran alrededor de dos operaciones básicas: **get( )** y **put( )**. Para poner un valor en un mapa se utiliza **put( )**, especificando la clave y el valor. Para obtener un valor se llama a **get( )** pasando la clave como argumento y el valor es devuelto.

Como hemos mencionado antes, los mapas no son colecciones pero se puede obtener una vista de colección de un mapa. Para hacer esto se puede utilizar el método **entrySet( )**. Este método devuelve una colección tipo **Set** que contiene los elementos del mapa. Para obtener una vista de colección de las claves, se utiliza el método **keySet( )**.

Método	Descripción
void clear( )	Quita todos los pares clave/valor del mapa que invoca.
boolean containsKey(Object *k*)	Devuelve **true** si el mapa que invoca contiene a *k* como clave. De lo contrario, devuelve **false**.
boolean containsValue(Object *v*)	Devuelve **true** si el mapa que invoca contiene a *v* como valor. De lo contrario devuelve **false**.
Set<Map.Entry<K, V>> entrySet( )	Devuelve una colección tipo **Set** que contiene las entradas del mapa. El conjunto contiene objetos de tipo **Map.Entry**. Este método proporciona una vista de conjunto del mapa que invoca.
boolean equals(Object *obj*)	Devuelve **true** si *obj* es un **Map** y contiene las mismas entradas que el que invoca. De lo contrario devuelve **false**.
V get(Object *k*)	Devuelve el valor asociado a la clave *k*. Devuelve **null** si la llave no se encuentra en el mapa.
int hashCode( )	Devuelve el código de dispersión del mapa que invoca.
boolean isEmpty( )	Devuelve **true** si el mapa que invoca está vacío. De lo contrario, devuelve **false**.
Set<K> keySet( )	Devuelve una colección tipo **Set** que contiene las claves en el mapa que invoca. Este método proporciona una vista de conjunto de las claves del mapa que invoca.
V put(K *k*, V *v*)	Coloca una entrada en el mapa que invoca. Sobrescribiendo cualquier valor previo asociado con la clave. La clave y el valor son *k* y *v*, respectivamente. Devuelve **null** si la clave no existía previamente. De lo contrario, devuelve el valor previamente vinculado a la clave.

TABLA 17-10 Los métodos definidos por **Map**

void putAll(Map<? Extends K, ? extends V> *m*)	Pone todas las entradas de *m* en este mapa.
V remove(Object *k*)	Quita la entrada cuya clave es igual a *k*.
int size( )	Devuelve el número de pares clave/valor en el mapa.
Collection<V> values( )	Devuelve una colección que contiene los valores en el mapa. Este método proporciona una vista de conjunto de los valores en el mapa.

TABLA 17-10 Los métodos definidos por **Map** *(continuación)*

Para conseguir una vista de colección de los valores, se utiliza **values( )**. Las vistas de colección son el medio por el que los mapas se integran en la estructura de colecciones.

La interfaz SortedMap

La interfaz **SortedMap** extiende **Map.SortedMap** asegura que las entradas se mantengan en orden ascendente acorde con el valor de sus claves. **SortedMap** es una interfaz genérica declarada como:

```
interface SortedMap<K,V>
```

Donde, **K** especifica el tipo de las claves y **V** especifica el tipo de los valores.

Los métodos declarados por **SortedMap** se resumen en la Tabla 17-11. Muchos métodos generan una excepción **NoSuchElementException** cuando no hay elementos en el mapa que invoca. Una excepción **ClassCastException** se genera cuando un objeto no es compatible con los elementos en el mapa. Una excepción **NullPointerException** se genera si se intenta utilizar un objeto **null** en una colección **Map** que no acepta valores **null**. Una excepción **IllegalArgumentException** se genera cuando se utiliza un argumento no válido.

Los mapas ordenados permiten la manipulación eficiente de *submapas* (en otras palabras, subconjuntos de mapas). Para obtener un submapa se utilizan los métodos **headMap( )**, **tailMap( )**, o **subMap( )**. Para obtener la primera clave del conjunto, se llama a **firstKey( )**. Para obtener la última se llama a **lastKey( )**.

Método	Descripción
Comparator<? super K> comparator( )	Devuelve el comparador del mapa ordenado que invoca. Si el mapa que invoca utiliza el orden natural, se devuelve **null**.
K firstKey( )	Devuelve la primera clave en el mapa que invoca.
SortedMap<K, V> headMap(K *fin*)	Devuelve un mapa ordenado con las entradas del mapa que tienen claves menores que *fin*.
K lastKey( )	Devuelve la última clave en el mapa que invoca.
SortedMap<K, V> subMap(K inicio, K *fin*)	Devuelve un mapa que contiene las entradas con claves mayores o iguales que inicio y menores que *fin*.
SortedMap<K, V> tailMap(K *inicio*)	Devuelve un mapa que contiene todas las entradas con claves mayores o iguales que *inicio*.

TABLA 17-11 Los métodos definidos por **SortedMap**

La interfaz NavigableMap

La interfaz **NavigableMap** fue añadida por Java SE 6. **NavigableMap** extiende a **SortedMap** y declara el comportamiento de un mapa que soporta la recuperación de entradas con base a la

búsqueda de coincidencias cercanas para una clave o un conjunto de claves. **NavigableMap** es una interfaz genérica declarada como:

```
interface NavigableMap<K,V>
```

Aquí, **K** especifica el tipo de las claves y **V** especifica el tipo de los valores asociados con las llaves. Adicionalmente a los métodos heredados de **SortedMap**, **NavigableMap** añade los métodos listados en la tabla 17-12. Muchos métodos generan excepciones de tipo **ClassCastException** cuando un objeto es incompatible con las llaves en el mapa. Una excepción **NullPointerException** se genera si se intenta utilizar un objeto **null** y claves **null** no están permitidas en el conjunto. Una excepción de tipo **IllegalArgumentException** se genera si se utiliza un argumento no válido.

Método	Descripción
Map.Entry<K, V> ceilingEntry(K *obj*)	Busca en el mapa por la clave *k* más pequeña, tal que *k* >= *obj*. Si esta clave es encontrada, la entrada es devuelta. En caso contrario se devuelve **null**.
K ceilingKey(K *obj*)	Busca en el mapa por la clave *k* más pequeña, tal que *k* >= *obj*. Si esta clave es encontrada, es devuelta. En caso contrario se devuelve **null**.
NavigableSet<K > descendingKeySet( )	Devuelve un **NavigableSet** que contiene las claves en el mapa que invoca en orden inverso. Esto es, devuelve una vista en conjunto de las llaves en orden inverso.
NavigableMap<K,V > descendingMap( )	Devuelve un **NavigableMap** que es el inverso del mapa que invoca.
Map.Entry<K, V>firstEntry( )	Devuelve la primera entrada en el mapa que invoca. Ésta es la entrada con la clave menor.
Map.Entry<K, V>floorEntry(K *obj*)	Busca en el mapa por la clave *k* mayor, tal que *k* < = *obj*. Si se encuentra una llave con estas características su entrada es devuelta. En caso contrario se devuelve **null**.
K floorKey(K *obj*)	Busca en el mapa por la clave *k* mayor, tal que *k* < = *obj*. Si se encuentra una llave con estas características, ésta es devuelta. En caso contrario se devuelve **null**.
NavigableMap<K, V> headMap(K *limiteSuperior*, boolean *i*)	Devuelve un **NavigableMap** que incluye todas las entradas del mapa que invoca que cuentan con una clave menor que el parámetro *limiteSuperior*. Si la variable *i* es **true**, entonces un elemento con el mismo valor que *limiteSuperior* sería incluido en el resultado.
Map.Entry<K, V>higherEntry (K *obj*)	Busca en el conjunto la clave *k* mayor, tal que *k* > *obj*. Si esta clave existe su entrada completa es devuelta. En caso contrario, se devuelve **null**.
K higherKey (K *obj*)	Busca en el conjunto la clave *k* mayor, tal que *k* > *obj*. Si esta clave existe es devuelta. En caso contrario, se devuelve **null**.
Map.Entry<K, V>lastEntry ( )	Devuelve la última entrada en el mapa. Ésta es la entrada con la clave mayor.
Map.Entry<K, V>lowerEntry (K *obj*)	Busca en el conjunto la clave *k* mayor, tal que *k* < *obj*. Si esta clave existe su entrada completa es devuelta. En caso contrario, se devuelve **null**.
K lowerKey (K *obj*)	Busca en el conjunto la clave *k* mayor, tal que *k* < *obj*. Si esta clave existe es devuelta. En caso contrario, se devuelve **null**.
NavigableSet<K> navigableKeySet( )	Devuelve un **NavigableSet** que contiene las claves del mapa que invoca.

TABLA 17-12 Los métodos definidos por **NavigableMap**

Método	Descripción
Map.Entry<K, V> pollFirstEntry( )	Devuelve y elimina la primera entrada. Debido a que el mapa está ordenado, la entrada devuelta y eliminada es aquella con el menor valor en su clave. Devuelve **null** si el mapa está vacío.
Map.Entry<K, V> pollLastEntry( )	Devuelve y elimina la última entrada. Debido a que el mapa está ordenado, la entrada devuelta y eliminada es aquella con el mayor valor en su clave. Devuelve **null** si el mapa está vacío.
NavigableMap<K, V> subMap(K *inferior*, boolean *ix*, K *superior*, boolean *sx*)	Devuelve un **NavigableMap** que incluye todas las entradas del mapa que invoca con claves mayores que el parámetro llamado inferior y menores que el parámetro llamado superior. Si *ix* es **true**, entonces los elementos iguales a inferior se incluyen en el resultado. Si *sx* es **true**, entonces los elementos iguales a superior se incluyen en el resultado.
NavigableMap <K, V> tailMap(K *inferior*, boolean *i*)	Devuelve un **NavigableMap** que incluye todas las entradas del mapa que invoca con claves mayores al parámetro llamado *inferior*. Si el parámetro *i* es **true**, entonces un elemento igual a *inferior* se incluye al resultado.

TABLA 17-12 Los métodos definidos por **NavigableMap** (*continuación*)

La interfaz Map.Entry

La interfaz **Map.Entry** permite trabajar con una entrada de mapa. Recuérdese que el método **entrySet( )** declarado por la interfaz **Map** devuelve una colección tipo **Set** que contiene las entradas del mapa. Cada uno de los elementos del conjunto es un objeto **Map.Entry. Map.Entry** es una interfaz genérica declarada como:

```
interface Map.Entry<K,V>
```

Donde, **K** especifica el tipo de claves, y **V** especifica el tipo de valores. La Tabla 17-13 resume los métodos declarados por **Map.Entry**. Varios de estos métodos generan excepciones.

Método	Descripción
Boolean equals (Object *obj*)	Devuelve **true** si *obj* es un **Map.Entry** cuyas claves y valores son iguales a aquellas del objeto que invoca.
K getKey( )	Devuelve la clave de la entrada que realiza la invocación.
V getValue( )	Devuelve el valor de la entrada que realiza la invocación.
int hashCode( )	Devuelve el código de dispersión de la entrada que realiza la invocación.
V setValue(V *v*)	Asigna el valor *v* a la entrada que invoca. Una excepción de tipo **ClassCastException** se genera si *v* no es el tipo correcto en el mapa. Una excepción de tipo **IllegalArgumentException** se genera si existe un problema con *v*. Una excepción de tipo **NullPointerException** se genera si *v* es **null** y el mapa no acepta claves null. Una excepción **UnsupportedOperationException** se genera si el mapa no puede ser modificado.

TABLA 17-13 Los métodos definidos por **Map.Entry**

Las clases Map

Existen varias clases que implementan las interfaces de mapas. Las clases que se pueden utilizar con mapas se resumen a continuación:

Clase	Descripción
AbstractMap	Implementa la mayor parte de la interfaz **Map**.
EnumMap	Extiende **AbstractMap** para utilizar claves tipo **enum**.
HashMap	Extiende **AbstractMap** para utilizar una tabla de dispersión.
TreeMap	Extiende **AbstractMap** para usar un árbol.
WeakHashMap	Extiende **AbstractMap** para utilizar una tabla de dispersión con claves débiles.
LinkedHashMap	Extiende **HashMap** para permitir iterar la colección en el orden de inserción.
IdentityHashMap	Extiende **AbstractMap** y utiliza igualdad basada en referencias para comparar documentos.

Nótese que **AbstractMap** es una superclase para las todas las implementaciones concretas. **WeakHashMap** implementa un mapa que utiliza "claves débiles", lo que permite que la memoria de un elemento de un mapa se recicle cuando su clave no se usa.

La clase HashMap

La clase **HashMap** extiende **AbstractMap** e implementa la interfaz **Map**. Utiliza una tabla de dispersión para almacenar el mapa. Esto permite que el tiempo de ejecución de operaciones básicas, como **get( )** y **put( )**, permanezca constante incluso para conjuntos grandes. **HashMap** es una clase genérica declarada como:

class HashMap <K,V>

Donde, **K** especifica el tipo de clave, y **V** especifica el tipo de valores.

Se definen los siguientes constructores para HashMap:

HashMap( )
HashMap(Map <? extends K, ? extends V > *m*)
HashMap(int *capacidad*)
HashMap(int *capacidad*, float *razonLlenado*)

La primera forma construye un mapa de dispersión por omisión. La segunda forma inicializa el mapa de dispersión utilizando los elementos de *m*. La tercera forma inicializa la capacidad del mapa a *capacidad*. La cuarta forma inicializa tanto la capacidad como la razón de llenado del mapa de dispersión usando sus argumentos. El significado de capacidad y razón de llenado es el mismo que para **HashSet**, anteriormente descrito. La capacidad por omisión es 16. La razón de llenado por omisión es 0.75.

HashMap implementa a **Map** y extiende **AbstractMap**. Esta clase no añade métodos propios.

Debe tenerse en cuenta que un mapa de dispersión *no* garantiza el orden de sus elementos. Por tanto, el orden en que se añaden elementos a un mapa de dispersión no es necesariamente el orden en que los lee un iterador.

El siguiente programa ilustra el uso de **HashMap**. El programa proyecta nombres a saldos de cuentas. Nótese como se obtiene y utiliza usa una vista de conjunto.

```
import java.util.*;

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

```
    // Crear un mapa de dispersión
    HashMap<String, Double> hm = new HashMap<String, Double> ();

    // Poner elementos en el mapa
    hm.put("Ken Bauer", new Double(3434.34));
    hm.put("Tom Smith", new Double(123 .22));
    hm.put("Jane Baker", new Double(1378.00));
    hm.put("Todd Hall", new Double(99.22));
    hm.put("Ralph Smith", new Double(-19.08));"

    // Obtener un conjunto con las entradas
    Set <Map.Entry <String, Double>> set = hm.entrySet();

    // Mostrar el conjunto
    for (Map.Entry<String, Double> me : set) {
      System.out.print(me.getKey() + ": ");
      System.out.println(me.getValue());
    }

    System.out.println( );

    // Depositar 1000 en la cuenta de Ken Bauer
    double balance = hm.get("Ken Bauer");
    hm.put("Ken Bauer", balance + 1000);

    System.out.println("Saldo actualizado de Ken Bauer: " +
      hm.get("Ken Bauer"));
  }
}
```

La salida de este programa es la siguiente aunque el orden puede variar:

```
Ralph Smith: -19.08
Todd Hall: 99.22
Ken Bauer: 3434.34
Jane Baker: 1378.0
Tom Smith: 123.22

Saldo actualizado de Ken Bauer: 4434.34
```

El programa comienza creando un mapa de dispersión y luego añade el mapeo entre nombres y saldos. Luego se muestran los contenidos del mapa utilizando una vista de conjunto, obtenida llamando a **entrySet( )**. Las claves y los valores se muestran llamando a los métodos **getKey( )** y **getValue( )** definidos por **Map.Entry**. Préstese especial atención en cómo se hace el ingreso en la cuenta de Ken Bauer. El método **put( )** remplaza automáticamente cualquier valor preexistente asociado a la clave especificada con el nuevo valor. Así, tras actualizar la cuenta de Ken Bauer, el mapa de dispersión sigue conteniendo sólo una cuenta para"Ken Bauer".

La clase TreeMap

La clase **TreeMap** extiende de **AbstractMap** e implementa la interfaz **NavigableMap**. Esta clase construye mapas almacenados en una estructura de datos tipo árbol. Un **TreeMap** proporciona un medio eficiente de almacenar pares clave/valor en orden, permitiendo una recuperación rápida. Debe notarse que, al contrario que un mapa de dispersión, un mapa en árbol garantiza que sus elementos se ordenan en orden ascendente de acuerdo a la clave. **TreeMap** es una clase genérica declarada como:

class TreeMap <K,V>

Donde, **K** especifica el tipo de las claves y **V** especifica el tipo de los valores.

TreeMap define los siguientes constructores:

TreeMap( )
TreeMap(Comparator<? super K> *comp*)
TreeMap(Map<? extends K, ? extends V> *m*)
TreeMap(SortedMap<K, ? extends V> *mo*)

La primera forma construye un mapa en árbol vacío que se ordenará utilizando el orden natural de sus claves. La segunda forma construye un mapa basado en árbol vacío, que se ordenará utilizando el **Comparator** *comp* (los comparadores se tratan más adelante en este capítulo). La tercera forma inicializa un mapa en árbol con las entradas de *m*, que se ordenarán utilizando el orden natural de sus claves. La cuarta forma inicializa un mapa en árbol con las entradas de *mo*, y que se ordenará en el mismo orden que *mo*.

TreeMap no define métodos adicionales propios, tiene los especificados por la interfaz **NavigableMap** y la clase **AbstractMap**.

El siguiente programa rehace el ejemplo anterior utilizando **TreeMap**:

```
import java.util.*;

class TreeMapDemo {
  public static void main{String args[]) {

    // Crear un mapa en árbol
    TreeMap<String, Double> tm = new TreeMap<String, Double>();

    // Poner elementos en el mapa
    tm.put("Ken Bauer", new Double (3434.34) ) ;
    tm.put("Tom Smith", new Double(123.22));
    tm.put("Jane Baker", new Double(1378.00));
    tm.put("Todd Hall", new Double(99.22));
    tm.put("Ralph Smith", new Double(-19.08));

    // Obtener un conjunto con las entradas
    Set <Map.Entry <String, Double>> set = tm.entrySet();

    // Mostrar los elementos
    for (Map.Entry<String, Double> me : set) {
      System.out.print(me.getKey() + ": ");
      System.out.println(me.getValue());
    }
    System.out.println();

    // Ingresar 1000 en la cuenta de Ken Bauer
    double balance = tm.get("Ken Bauer");
    tm.put("Ken Bauer", balance + 1000);

    System.out.println("Nuevo saldo de Ken Bauer: " +
      tm.get ("Ken Bauer"));
  }
}
```

La siguiente es la salida del programa:

```
Jane Baker: 1378.0
Ken Bauer: 3434.34
Ralph Smith: -19.08
Todd Hall: 99.22
Toro Smith: 123.22
```

```
Nuevo saldo de Ken Bauer: 4434.34
```

Nótese que **TreeMap** ordena las claves. Sin embargo, en este caso están ordenadas por nombre en vez de por apellido. Se puede alterar este comportamiento especificando un comparador cuando se crea un mapa, como se describe más adelante.

La clase LinkedHashMap

La clase **LinkedHashMap** extiende **HashMap**. Esta clase mantiene una lista enlazada de las entradas en el mapa, en el orden en que fueron insertadas. Esto permite el recorrido del mapa. Esto es, cuando se recorre una vista de la colección **LinkedHashMap**, los elementos serán devueltos en el orden en el cual fueron insertados. Se puede crear también un **LinkedHashMap** que devuelva los elementos en el orden en el cual fueron ingresados por última vez. **LinkedHashMap** es una clase genérica declarada como:

```
class LinkedHashMap <K,V>
```

Donde, **K** especifica el tipo de las claves, y **V** especifica el tipo de los valores.

LinkedHashMap define los siguientes constructores:
LinkedHashMap( )
LinkedHashMap(Map<? extends K, ? extends V> *m*)
LinkedHashMap(int *capacidad*)
LinkedHashMap(int *capacidad*, float *razonLlenado*)
LinkedHashMap(int *capacidad*, float *razonLlenado*, boolean *orden*)

La primera forma construye un **LinkedHashMap** por omisión. La segunda forma inicializa al **LinkedHashMap** con los elementos de *m*. La tercera forma inicializa la capacidad. La cuarta forma inicializa tanto la capacidad como la razón de llenado. El significado de capacidad y razón de llenado son los mismos que para **HashMap**. La capacidad por omisión es 16. La razón de llenado por omisión es 0.75. La última forma nos permite especificar si los elementos serán almacenados en la lista enlazada en orden de inserción o en el orden de último acceso. Si *orden* es **true**, se emplea un ordenamiento basado en el número de accesos a los elementos. Si *orden* es **false** se emplea ordenamiento acorde al orden de inserción de los elementos en la colección.

LinkedHashMap sólo añade un método a los definidos por **HashMap**. Este método es **removeEldestEntry( )** que está declarado como:

```
protected boolean removeEldestEntry(Map.Entry<K,V> e)
```

Este método es llamado por el método **put( )** y **putAll( )**. La entrada más antigua está dada por *e*. Por omisión, este método devuelve **false** y no realiza ninguna acción. Sin embargo, si sobrescribimos el método, podríamos tenerlo disponible en nuestros mapas. Para hacer esto, nuestro método debe regresar **true**, para identificarlo del original.

La clase IdentityHashMap

IdentityHashMap extiende de **AbstractMap** e implementa la interfaz **Map**. Esta clase es similar a **HashMap** excepto que ésta utiliza comparación de igualdad por referencias. **IdentityHashMap** es una clase genérica declarada como:

```
class IdentityHashMap <K,V>
```

Aquí, **K** especifica el tipo de las claves, y **V** especifica el tipo de los valores. La documentación del API de Java especifica explícitamente que **IdentityHashMap** no es de uso general.

La Clase EnumMap

EnumMap extiende **AbstractMap** e implementa **Map**. Esta clase es específica para utilizarse con claves de tipo enumerado. Es una clase genérica declarada como:

Class EnumMap <K extends Enum<K>, V>

Donde, **K** especifica el tipo de las claves, y **V** especifica el tipo de los valores. Nótese que **K** debe extender **Enum<K>**, lo cual refuerza el requerimiento para que las claves sean de tipo enumerado.

EnumMap define los siguientes constructores:

EnumMap (Class<K> *tipok*)
EnumMap (Map<K, ? extends V> *m*)
EnumMap (EnumMap<K, ? extends V> *em*)

El primer constructor crea un **EnumMap** vacío de tipo *tipok*. El segundo crea un **EnumMap** que contiene las mismas entradas que *m*. El tercero crea un **EnumMap** inicializado con los valores en *em*.

EnumMap no define métodos propios.

Comparadores

Tanto **TreeSet** como **TreeMap** almacenan elementos ordenadamente. Es el comparador el que define de forma precisa qué significa "ordenadamente". Por omisión, estas clases almacenan sus elementos utilizando lo que Java llama "orden natural", que normalmente es el orden que se puede esperar. (A antes que B, 1 antes que 2, etc.). Para ordenar los elementos de otra forma se especifica un objeto **Comparator** al construir el conjunto o mapa. Hacer esto permite controlar exactamente cómo se ordenan los elementos dentro de colecciones y mapas ordenados.

Comparator es una interfaz genérica declarada como:

interface Comparator<T>

Donde, **T** especifica el tipo de objetos que serán comparados.

La interfaz **Comparator** define dos métodos: **compare( )** y **equals( )**. El método **compare( )** aquí mostrado, compara dos elementos en cuanto a su orden:

int compare(T *objl*, T *obj2*)

objl y *obj2* son los objetos que han de compararse. Este método devuelve cero si los objetos son iguales, un valor positivo si *objl* es mayor que *obj2*, y en cualquier otro caso, un valor negativo. El método puede producir una excepción **ClassCastException** si los tipos de objetos no son compatibles. Sobrescribiendo **compare( )** se puede alterar el modo en que los objetos se ordenan. Por ejemplo, para ordenar en orden inverso, se puede crear un comparador que invierta el resultado de una comparación.

El método **equals( )**, mostrado aquí, comprueba si un objeto es igual al comparador que invoca:

boolean equals(Object *obj*)

Aquí, *obj* es el objeto cuya igualdad se comprueba. El método devuelve **true** si *obj* y el objeto que invoca son ambos objetos **Comparator** y utilizan la misma ordenación. De lo contrario, devuelve **false**. Sobrescribir **equals( )** es innecesario, y la mayoría de los comparadores simples no lo harán.

Uso de un comparador

El siguiente ejemplo ilustra el poder de un comparador personalizado. Implementa el método **compare( )** para que opere en orden inverso. Así hace que un TreeSet se almacene en orden inverso.

```
// Uso de un comparador personalizado
import java.util.*;

// Un comparador invertido para cadenas.
class MyComp implements Comparator<String> {
  public int compare(Object a, Object b)
    String aStr, bStr;

    aStr = a;
    bStr = b;

    // invertir la comparación
    return bStr.compareTo(aStr);
  }

  // en caso de igualdad no hace falta sobrescribir
}

class CompDemo{
  public static void main(String args[]) {
    // Crear un TreeSet
    TreeSet<String> ts = new TreeSet<String>(new MyComp());

    // Añadir elementos al árbol
    ts.add("C");
    ts.add("A");
    ts.add("B");
    ts.add("E");
    ts.add("F");
    ts.add("D");

    // Mostrar los elementos
    for (String elemento : ts) {
      System.out.print(elemento + " ");

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

Como se muestra en la salida, el árbol se almacena ahora en orden inverso:

```
F E D C B A
```

Obsérvese bien la clase **MyComp**, que implementa **Comparator** y sobrescribe al método **compare( )**. Como ya se ha explicado, sobrescribir **equals( )** no es necesario ni común. Dentro de **compare( )**, el método **compareTo( )** de la clase String compara dos cadenas. Sin embargo, **bStr**—no **aStr**—llama a **compareTo( )**. Esto hace que el resultado de la comparación se invierta.

Como un ejemplo más práctico, el siguiente programa es una versión actualizada del programa **TreeMap** de la sección anterior, que almacena saldos de cuentas. En la versión anterior, las cuentas se ordenaron por nombre, no por apellido. El siguiente programa ordena las cuentas por apellido. Para hacerlo, usa un comparador que compara el apellido de cada cuenta. El resultado es que el mapa se ordena por apellido.

```
// Uso de un comparador para ordenar las cuentas por apellido.
import java.util.*;

// Compara las últimas palabras completas de dos cadenas.
class TComp implements Comparator<String> {
  public int compare(Object a, Object b) {
    int i, j, k;
    String aStr, bStr;

    aStr = a;
    bStr = b;

    // encontrar el índice del comienzo del apellido
    i = aStr.lastIndexOf(' ');
    j = bStr.lastIndexOf(' ');

    k = aStr.substring(i) .compareTo(bStr.substring(j));
    if(k==0) // si los apellidos coinciden, comprobar el nombre entero
      return aStr.compareTo(bStr);
    else
      return k;
  }

  // no es necesario sobrescribir el método equals
}

class TreeMapDemo2 {
  public static void main(String args[]) {
    // Crear un mapa nuevo
    TreeMap<String, Double> tm = new TreeMapp<String, Double> (new TComp());

    // Poner elementos en el mapa
    tm.put("Ken Bauer", new Double(3434.34));
    tm.put("Tom Smith", new Double(123.22));
    tm.put("Jane Baker", new Double(1378.00));
    tm.put("Todd Hall", new Double(99.22));
    tm.put("Ralph Smith", new Double(-19.08));

    // Obtener un conjunto con las entradas
    Set>Map.Entry<String, Double>> set = tm.entrySet();

    // Mostrar los elementos
    for (Map.Entry<String, Double> me : set ) {
      System.out.print(me.getKey() + ": ");
      System.out.println(me.getValue());
    }
    System.out.println();

    // Ingresar 1000 en la cuenta de Ken Bauer
    double balance = tm.get ("Ken Bauer");
    tm.put("Ken Bauer", balance + 1000);

    System.out.println("Nuevo saldo en la cuenta de Ken Bauer: " +
      tm.get("Ken Bauer"));
  }
}
```

Aquí está la salida; nótese que las cuentas se ordenaron ahora por apellido:

```
Jane Baker: 1378.0
Todd Hall: 99.22
```

```
Ken Bauer: 3434.34
Ralph Smith: -19.08
Tom Smith: 123.22

Nuevo saldo en la cuenta de Ken Bauer: 4434.34
```

La clase comparador **Tcomp** compara dos cadenas que contienen nombres y apellidos. Lo hace comparando primero los apellidos. Para ello, encuentra el índice del último espacio en cada cadena y luego compara las subcadenas de cada elemento que comienzan en ese punto. En los casos en que los apellidos son equivalentes, se comparan entonces los nombres. Esto produce un mapa en árbol que está ordenado por apellidos, y cuando éste coincide, por nombre. Se puede ver esto en el hecho de que Ralph Smith venga antes que Tom Smith en la salida.

Los algoritmos de la estructura de colecciones

La estructura de colecciones define diversos algoritmos que se pueden aplicar a colecciones y mapas. Estos algoritmos están definidos como métodos estáticos dentro de la clase **Collections**. Se resumen en la Tabla 17-14. Como se explicó antes, a partir de JDK 5 todos los algoritmos han sido mejorados para utilizar tipos parametrizados. Aunque la nueva sintaxis puede parecer intimidante al inicio, los algoritmos son tan sencillos de utilizar como antes de que se incluyeran los tipos parametrizados. Sólo que ahora cuentan con mayor seguridad en el manejo de tipos.

Método	Descripción
static <T> boolean addAll(Collection <? super T> *c*, T … *elements*)	Inserta los *elementos* especificados por *e* en la colección especificada por *c*. Devuelve **true** si los elementos fueron añadidos y **false** en caso contrario.
static <T> Queue<T> asLifoQueue(Dequeue<T> *c*)	Devuelve una vista "último en entrar – primero en salir" de *c* (añadido en Java SE 6).
static <T> int binarySearch(List<? extends T> *lista*, T *valor*, Comparator<? Super T> *c*)	Busca el dato dado en la variable *valor* en la colección lista la cual está ordenada acorde con *c*. Devuelve la posición de *valor* en la *lista* o un valor negativo si *valor* no se encuentra en la *lista* dada como parámetro.
static <T> int binarySearch(List <? extends Comparable<? super T>> *lista*, T *valor*)	Busca *valor* en *lista*. La lista debe ser ordenada. Devuelve la posición de *valor* en *lista*, o un valor negativo si *valor* no se encuentra.
static <E> Collection<E> checkedCollection(Collection<E> *c*, Class<E> *t*)	Devuelve en tiempo de ejecución una vista de la colección con seguridad de tipos. Un intento de insertar un elemento no compatible genera una excepción de tipo **ClassCastException**.
static <E> List<E> checkedList(List<E> *c*, Class<E> *t*)	Devuelve en tiempo de ejecución una vista de la **lista** con seguridad de tipos. Un intento de insertar un elemento no compatible genera una excepción de tipo **ClassCastException**.
static <K, V> Map<K, V> checkedMap(Map<K, V> *c*, Class<K> *claveT*, Class<V> *valorT*)	Devuelve en tiempo de ejecución una vista del **mapa** con seguridad de tipos. Un intento de insertar un elemento no compatible genera una excepción de tipo **ClassCastException**.

TABLA 17-14 Los algoritmos definidos por **Collections**

Método	Descripción
static <E> List<E> checkedSet(Set<E> c, Class<E> *t*)	Devuelve en tiempo de ejecución una vista del conjunto con seguridad de tipos. Un intento de insertar un elemento no compatible genera una excepción de tipo **ClassCastException**.
static <K, V> SortedMap<K, V> checkedSortedMap(SortedMap<K, V> *c*, Class<K> *claveT*, Class<V> *valorT*)	Devuelve en tiempo de ejecución una vista del **mapa ordenado** con seguridad de tipos. Un intento de insertar un elemento no compatible genera una excepción de tipo **ClassCastException**.
static <E> SortedSet<E> checkedSortedSet(SortedSet<E> *c*, Class<E> *t*)	Devuelve en tiempo de ejecución una vista del **conjunto ordenado** con seguridad de tipos. Un intento de insertar un elemento no compatible genera una excepción de tipo ClassCastException.
static <T> void copy(List<? super T> *lista1*, List<? super T> *lista2*)	Copia los elementos de *lista2* en *lista1*.
static boolean disjoint(Collection<?> *a*, Collection<?> *b*)	Compara los elementos de *a* con los elementos de *b*. Devuelve **verdades** si las dos colecciones no contienen elementos en común (esto es, las colecciones contienen conjuntos disjuntos de elementos). En otro caso devuelve **falso**.
static <T> List<T> emptyList( )	Devuelve un objeto **List** que representa una lista inmutable y vacía.
static <K, V> Map<K, V> emptyMap( )	Devuelve un objeto **Map** que representa un mapa inmutable y vacío.
static <T> Set<T> emptySet( )	Devuelve un objeto **Set** que representa un conjunto inmutable y vacío.
static <T> Enumeration<T> enumeration (Collection<T> *c*)	Devuelve una enumeración sobre *c*. (véase "La interfaz de enumeración", más adelante en este capítulo).
Static <T> void fill(List<? super T> lista, T *obj*)	Asigna *obj* a cada elemento de *lista*.
static int frequency(Collection<?> c, Object *obj*)	Cuenta el número de ocurrencias de *obj* en *c* y devuelve el resultado.
static int indexOfSubList(List<?> *lista*, List<?> *sublista*)	Busca en *lista* la primer ocurrencia de *sublista*. Devuelve el índice de la primera ocurrencia o –1 si ninguna ocurrencia es encontrada.
static int lastIndexOfSubList(List<?> *lista*, List<?> *sublista*)	Busca en lista la última ocurrencia de *sublista*. Devuelve el índice de la primera ocurrencia o –1 si ninguna ocurrencia es encontrada.
static <T> ArrayList<T> list (Enumeration<T> *enum*)	Devuelve un **ArrayList** que contiene los elementos de *enum*.
static <T> T max (Collection <? extends T> *c*, Comparator<? super T> *comparador*)	Devuelve el elemento máximo en *c* según lo determina *comparador*.
static <T extends Object& Comparable<? super T>> T max(Collection<? extends T> *c*)	Devuelve el máximo elemento de *c* según lo determina el orden natural. La colección no necesita ser ordenada.
static <T> T min(Collection<? extends T> *c*, Comparator<? super T> *comparador*)	Devuelve el elemento mínimo de *c* según lo determina *comparador*. La colección no necesita ser ordenada.

TABLA 17-14 Los algoritmos definidos por **Collections** *(continuación)*

Método	Descripción
static <T extends Object & Comparable<? super T>> T min(Collection<? extends T> *c*)	Devuelve el mínimo elemento de *c* según el orden natural. La colección no requiere estar ordenada.
static <T> List<T> nCopies(int *num*, T *obj*)	Devuelve *num* copias de *obj* en una lista inmutable. *num* debe ser mayor o igual a cero.
static <E> Set<E> newSetFromMap(Map<E, Boolean> *m*)	Crea y devuelve un conjunto a partir del mapa especificado en *m*, el cual debe estar vacío cuando el método sea llamado. (añadido por Java SE 6).
static <T> boolean replaceAll(List<T> *lista*, T *anterior,* T *nuevo*)	Remplaza todas las ocurrencias de anterior con nuevo en lista. Devuelve **true** si al menos un remplazo ocurre. Devuelve **false** en caso contrario.
static void reverse(List<T> *lista*)	Invierte la secuencia en la *lista*.
static <T> Comparator<T> reverseOrder(Comparator<T> *comp*)	Devuelve un comparador inverso basado en aquel que se pase como argumento. Esto es, el comparador devuelto invierte el resultado de una comparación hecha con *comp*.
static <T> Comparator<T> reverseOrder( )	Devuelve un comparador inverso, el cual es un comparador que invierte el resultado de una comparación entre dos elementos.
static void rotate(List<T> *lista*, int *n*)	Rota *lista*, *n* lugares a la derecha. Para rotar a la izquierda utilice un valor negativo para *n*.
static void shuffle(List<T> *lista*, Random *r*)	Mezcla (aleatoriamente) los elementos de *lista* utilizando *r* como fuente de números aleatorios.
static void shuffle(List<T> *lista*)	Mezcla (aleatoriamente) los elementos de lista.
static <T> Set<T> singleton(T *obj*)	Devuelve *obj* como conjunto inmutable. Ésta es una manera fácil de convertir un objeto único en un conjunto.
static <T> List<T> singletonList(T *obj*)	Devuelve *obj* como lista inmutable. Éste es un modo fácil de convertir un objeto único en una lista.
static <K, V> Map <K, V> singletonMap(K *k*, V *v*)	Devuelve el par clave/valor (*k, v*) como un mapa inmutable. Éste es un modo fácil de convertir un par clave/valor único en un mapa.
static <T> void sort(List<T> *lista*, Comparator<? super T> *comp*)	Ordena los elementos de *lista* según lo determine *comp*.
static <T extends Comparable<? super T>> void sort(List<T> *lista*)	Ordena los elementos de *lista* acorde a su orden natural.
static void swap(List<?> *lista*, int *idx1*, int *idx2*)	Intercambia los elementos en lista en los índices especificados por *idx1* e *idx2*.
static <T> Collection<T> synchronizedCollection(Collection<T> *c*)	Devuelve una colección con seguridad para trabajar con hilos a partir de colección común.
static <T> List<T> synchronizedList(List<T> *lista*)	Devuelve una *lista* con seguridad para trabajar con hilos a partir de una lista común.
static <K, V> Map<K, V> synchronizedMap(Map<K, V> *m*)	Devuelve un mapa con seguridad para trabajar con hilos a partir de *m*.
static <T> Set<T> synchronizedSet(Set<T> s)	Devuelve un conjunto con seguridad para trabajar con hilos a partir del conjunto s.

TABLA 17-14 Los algoritmos definidos por **Collections** *(continuación)*

Método	Descripción
static <K, V> SortedMap<K, V> synchronizedSortedMap(SortedMap<K, V> *sm*)	Devuelve un mapa ordenado con seguridad para trabajar con hilos a partir del mapa ordenado *sm*.
Static <T> SortedSet<T> synchronizedSortedSet(SortedSet<T> ss)	Devuelve un conjunto con seguridad para trabajar con hilos a partir del conjunto ordenado ss.
Static <T> Collection<T> unmodifiableCollection(Collection<? extends T> *c*)	Devuelve una colección inmodificable a partir de *c*.
static <T> List<T> unmodifiableList(List<? extends T> *lista*)	Devuelve una lista inmodificable a partir de *lista*.
static <K, V> Map<K, V> unmodifiableMap (Map<? extends K, ? extends V> *m*)	Devuelve un mapa inmodificable a partir de *m*.
static <T> Set<T> unmodifiableSet(Set<? extends T> *s*)	Devuelve un conjunto inmodificable a partir de s.
static <K, V> SortedMap<K, V> unmodifiableSortedMap(SortedMap<K, ? extends V> *sm*)	Devuelve un mapa ordenado inmodificable a partir de *sm*.
static <T> SortedSet<T> unmodifiableSortedSet(SortedSet<T> *c*)	Devuelve un conjunto ordenado inmodificable a partir de *c*.

TABLA 17-14 Los algoritmos definidos por **Collections** *(continuación)*

Varios de los métodos pueden producir la excepción **ClassCastException**, que ocurre cuando se intenta comparar tipos incompatibles, o una **UnsupportedOperationException**, que se da cuando se intenta modificar una colección inmodificable. Diversas excepciones se pueden producir en los diferentes métodos.

Los métodos **check( )** ameritan especial atención, por ejemplo **checkedCollection( )**, el cual regresa lo que la documentación del API denomina"vista dinámica con seguridad de tipos"de una colección. Esta vista es una referencia a la colección que monitorea la inserción en la colección por compatibilidad de tipos en tiempo de ejecución. Un intento de insertar un elemento incompatible causará una excepción de tipo **ClassCastException**. Utilizar una vista es especialmente útil durante la depuración debido a que asegura que la colección siempre contenga elementos válidos. Métodos como **checkedSet( )**, **checkedList( )**, **checkedMap( )** y otros intervienen en esta actividad. Estos métodos obtienen una vista con seguridad de tipos para la colección indicada.

Nótese que se utilizan varios métodos, como **synchronizedList( )** y **synchronizedSet( )**, para obtener copias sincronizadas (con *seguridad para trabajar con hilos*) de las diversas colecciones. Como ya se ha dicho, ninguna de las implementaciones de colecciones estándar está sincronizada. Para proporcionar sincronización hay que utilizar los algoritmos de sincronización. Una nota más: los iteradores sobre colecciones sincronizadas deben utilizarse dentro de bloques **sincronizados** (recuerde la palabra clave **synchronized**).

El conjunto de métodos que comienza con **unmodifiable** devuelve vistas de las diversas colecciones que no se pueden modificar. Éstas son útiles cuando se quiere otorgar a un proceso la capacidad de lectura —pero no escritura— sobre una colección.

Collections define tres variables estáticas: **EMPTY_SET**, **EMPTY_LIST**, y **EMPTY_MAP**. Las tres son inmutables.

El siguiente programa ilustra algunos de los algoritmos. Crea e inicializa una lista enlazada. El método **reverseOrder( )** devuelve un **Comparator** que invierte la comparación de objetos **Integer**. Los elementos de la lista se ordenan de acuerdo con su comparador y luego se muestran. A continuación la lista se mezcla aleatoriamente llamando a **shuffle( )**, y se muestran sus valores mínimo y máximo.

```
// Ejemplo de varios algoritmos.
import java.util.*;

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

    // Crear e inicializar una lista enlazada
    LinkedList<Integer> l1 = new LinkedList<Integer>();
    l1.add(-8);
    l1.add(20);
    l1.add(-20);
    l1.add(8);

    // Crear un comparador de orden inverso
    Comparator<Integer> r = Collections.reverseOrder();

    // Ordenar la lista utilizando el comparador
    Collections.sort(l1, r);

    System.out.print("Lista en orden inverso: ");
    for (int i : l1)
      System.out.print(i + " ");

    System.out.println();

    // Mezclar la lista
    Collections.shuffle(l1);

    // Mostrar la lista mezclada aleatoriamente
    System.out.print("Lista mezclada al azar: ");
    for (int i : l1)
      System.out.print(i + " ");

    System.out.println();

    System.out.println("Mínimo: " + Collections.min(l1));
    System.out.println("Máximo: " + Collections.max(l1));
  }
}
```

La salida de este programa se muestra a continuación:

```
Lista en orden inverso: 20 8 -8 -20
Lista mezclada al azar: 20 -20 8 -8
Mínimo: -20
Máximo: 20
```

Nótese que **min( )** y **max( )** operan sobre la lista después de haber sido mezclada. Ninguno requiere una lista ordenada para su funcionamiento.

Arrays

La clase **Arrays** proporciona diversos métodos útiles para trabajar con arreglos. Aunque técnicamente estos métodos no son parte de la estructura de colecciones, ayudan a unir colecciones y arreglos. En esta sección examinaremos los métodos definidos por **Arrays**.

El método **asList( )** devuelve una **Lista** a partir de un arreglo específico. En otras palabras, tanto la lista como el arreglo se refieren a la misma ubicación. Tiene la siguiente forma:

static <T> List asList(T… *arreglo*)

Aquí, *arreglo* es el arreglo que contiene los datos.

El método **binarySearch( )** utiliza una búsqueda binaria para encontrar un valor especificado. Este método se debe aplicar a arreglos ordenados. Éstas son algunas de sus formas. (Java SE 6 añade muchos otros).

static int binarySearch(byte[ ] *arreglo*, byte *valor*)
static int binarySearch(char[ ] *arreglo*, char *valor*)
static int binarySearch(double[ ] *arreglo*, double *valor*)
static int binarySearch(float[ ] *arreglo*, float *valor*)
static int binarySearch(int[ ] *arreglo*, int *valor*)
static int binarySearch(long[ ] *arreglo*, long *valor*)
static int binarySearch(short[ ] *arreglo*, short *valor*)
static int binarySearch(Object[ ] *arreglo*, Object *valor*)
static int binarySearch(T[ ] *arreglo*, T *valor*, Comparator<? super T> *c*)

Aquí, *arreglo* es el arreglo en el que se buscará, y *valor* es el valor que se ha de localizar. Las dos últimas formas producen una excepción **ClassCastException** si *arreglo* contiene elementos que no se pueden comparar (por ejemplo, **Double** y **StringBuffer**), o si *valor* no es compatible con los tipos en *arreglo*. En la última forma, el **comparador** *c* se utiliza para determinar el orden de los elementos en el *arreglo*. En todos los casos, si *valor* existe en *arreglo*, se devuelve el índice del elemento. De lo contrario, se devuelve un valor negativo.

El método **copyOf( )** fue añadido por Java SE 6. Este método devuelve una copia de un arreglo y tiene las siguientes formas:

static boolean[ ] copyOf(boolean[ ] *fuente*, int *t*)
static byte[ ] copyOf(byte[ ] *fuente*, int *t*)
static char[ ] copyOf(char[ ] *fuente*, int *t*)
static double[ ] copyOf(double[ ] *fuente*, int *t*)
static float[ ] copyOf(float[ ] *fuente*, int *t*)
static int[ ] copyOf(int[ ] *fuente*, int *t*)
static long[ ] copyOf(long[ ] *fuente*, int *t*)
static short[ ] copyOf(short[ ] *fuente*, int *t*)
static <T> T[ ] copyOf(T[ ] *fuente*, int *t*)
static <T, U> T[ ] copyOf(U[ ] *fuente*, int *t*, Class<? extends T[ ]> *resultado*)

El arreglo original está especificado en la variable *fuente*, y el tamaño de la copia está especificado por *t*. Si la copia es mayor a *fuente*, entonces la copia es rellenada con ceros (para arreglos numéricos), con **null** (para arreglos de objetos), con **false** (para arreglos boolean). Si la copia es más pequeña que *fuente*, entonces la copia es truncada. En la última forma, el tipo de *resultado* es el tipo del arreglo devuelto. Si *t* es negativo se genera una excepción

de tipo **NegativeArraySizeException**. Si *fuente* es **null** se genera una excepción de tipo **NullPointerException**. Si *resultado* es incompatible con el tipo de *fuente*, se genera una excepción de tipo **ArrayStoreException**.

El método **copyOfRange( )** fue también añadido por Java SE 6. Este devuelve una copia de un rango de valores del arreglo. Este método tiene las siguientes formas:

```
static boolean[ ] copyOfRange(boolean[ ] fuente, int inicio, int fin)
static byte[ ] copyOfRange(byte[ ] fuente, int inicio, int fin)
static char[ ] copyOfRange(char[ ] fuente, int inicio, int fin)
static double[ ] copyOfRange(double[ ] fuente, int inicio, int fin)
static float[ ] copyOfRange(float[ ] fuente, int inicio, int fin)
static int[ ] copyOfRange(int[ ] fuente, int inicio, int fin)
static long[ ] copyOfRange(long[ ] fuente, int inicio, int fin)
static short[ ] copyOfRange(short[ ] fuente, int inicio, int fin)
static <T> T[ ] copyOfRange(T[ ] fuente, int inicio, int fin)
static <T, U> T[ ] copyOfRange(U[ ] fuente, int inicio, int fin, Class<? extends T[ ]> resultado)
```

El arreglo original está especificado por *fuente*. El rango a copiar se especifica por los índices pasados por los argumentos *inicio* y *fin*. El rango va de *inicio* a *fin* – 1. Si el rango es mayor que *fuente*, entonces la copia es rellenada con ceros (para arreglos numéricos), con **null** (para arreglos de objetos), con **false** (para arreglos boolean). En la última forma, el tipo de *resultado* es el tipo del arreglo devuelto. Si *inicio* es negativo o mayor que la longitud de *fuente*, se genera una excepción de tipo **ArrayIndexOutOfBoundsException**. Si inicio es mayor que fin, se genera una excepción de tipo **IllegalArgumentException**. Si *fuente* es **null**, se genera una excepción **NullPointerException**. Si *resultado* es incompatible con el tipo de *fuente*, se genera una excepción de tipo **ArrayStoreException**.

El método **equals( )** devuelve **true** si dos arreglos son equivalentes. De lo contrario, devuelve **false**. El método **equals( )** tiene las siguientes formas:

```
static boolean equals(boolean arreglol[ ], boolean arreglo2[ ])
static boolean equals(byte arreglol [ ], byte arreglo2[ ])
static boolean equals(char arreglol[ ], char arreglo2[ ])
static boolean equals(double arreglol[ ], double arreglo2[ ])
static boolean equals(float arreglol[ ], float arreglo2[ ])
static boolean equals(int arreglol[ ], int arreglo2[ ])
static boolean equals(long arreglol[ ], long arreglo2[ ])
static boolean equals(short arreglol[ ], short arreglo2[ ])
static boolean equals(Object arreglol[ ], Object arreglo2[ ])
```

Aquí, *arreglol* y *arreglo2* son los dos arreglos cuya igualdad se comprueba.

El método **deepEquals( )** puede utilizarse para determinar si dos arreglos, que pueden contener arreglos anidados, son iguales. El método tiene la siguiente declaración:

```
static boolean deepEquals(Object[ ] a, Object[ ] b)
```

Devuelve **true** si los arreglos pasados en *a* y *b* contienen los mismos elementos. Si *a* y *b* contienen arreglos anidados, entonces el contenido de los arreglos anidados también es revisado. Devuelve **false** si los arreglos, o sus arreglos anidados, son diferentes.

El método **fill( )** asigna un valor a todos los elementos de un arreglo. En otras palabras, llena un arreglo con un valor especificado. El método **fill( )** tiene dos versiones. La primera, que tiene las siguientes formas, llena el arreglo completo:

static void fill(boolean *arreglo*[ ], boolean *valor*)
static void fill(byte *arreglo*[ ], byte *valor*)
static void fill(char *arreglo*[ ], char *valor*)
static void fill(double *arreglo*[ ], double *valor*)
static void fill(float *arreglo*[ ], float *valor*)
static void fill(int *arreglo*[ ], int *valor*)
static void fill(long *arreglo*[ ], long *valor*)
static void fill(short *arreglo*[ ], short *valor*)
static void fill(Object *arreglo*[ ], Object *valor*)

Aquí, *valor* se asigna a todos los elementos de *arreglo*.

La segunda versión del método **fill( )** asigna un valor a un subconjunto de un arreglo. Sus formas se muestran aquí:

static void fill(boolean *arreglo*[ ], int *inicio*, int *fin*, boolean *valor*)
static void fill(byte *arreglo*[ ], int *inicio*, int *fin*, byte *valor*)
static void fill(char *arreglo*[ ], int *inicio*, int *fin*, char *valor*)
static void fill(double *arreglo*[ ], int *inicio*, int *fin*, double *valor*)
static void fill(float *arreglo*[ ], int *inicio*, int *fin*, float *valor*)
static void fill(int *arreglo*[ ], int *inicio*, int *fin*, int *valor*)
static void fill(long *arreglo*[ ], int *inicio*, int *fin*, long *valor*)
static void fill(short *arreglo*[ ], int *inicio*, int *fin*, short *valor*)
static void fill(Object *arreglo*[ ], int *inicio*, int *fin*, Object *valor*)

Aquí, *valor* se asigna a los elementos de *arreglo* desde la posición *inicio* hasta la posición *fin*-1. Estos métodos pueden producir una excepción **IllegalArgumentException** si inicio es mayor que fin, o una excepción **ArrayIndexOutOfBoundException** si *inicio* o *fin* están fuera de los límites.

El **método sort( )** ordena un arreglo en orden ascendente. El método **sort( )** tiene dos versiones. La primera versión, mostrada aquí, ordena el arreglo completo:

static void sort(byte *arreglo*[ ])
static void sort(char *arreglo*[ ])
static void sort(double *arreglo*[ ])
static void sort(float *arreglo*[ ])
static void sort(int *arreglo*[ ])
static void sort(long *arreglo*[ ])
static void sort(short *arreglo*[ ])
static void sort(Object *arreglo*[ ])
static <T> void sort(T *arreglo*[ ], Comparator<? super T> *c*)

Aquí, *arreglo* es el arreglo que se ha de ordenar. En la última forma, *c* es un comparador que se utiliza para ordenar los elementos de *arreglo*. Las últimas dos formas pueden producir una excepción de tipo ClassCastException si los elementos del arreglo que se quiere ordenar no son comparables.

La segunda versión de **sort( )** permite especificar un rango que se quiere ordenar dentro de un arreglo. Son sus formas son:

static void sort(byte *arreglo*[ ], int *inicio*, int *fin*)
static void sort(char *arreglo*[ ], int *inicio*, int *fin*)
static void sort(double *arregl*

static void sort(float *arreglo*[ ], int *inicio*, int *fin*)
static void sort(int *arreglo*[ ], int *inicio*, int *fin*)
static void sort(long *arreglo*[ ], int *inicio*, int *fin*)
static void sort(short *arreglo*[ ], int *inicio*, int *fin*)
static void sort(Object *arreglo*[ ], int *inicio*, int *fin*)
static <T> void sort(T *arreglo*[ ], int *inicio*, int *fin*, Comparator<? super T> *c*)

Aquí se ordena el rango de arreglo desde *inicio* hasta *fin*-l. En la última forma, *c* es un **Comparator** que se utiliza para ordenar los elementos de *arreglo*. Todos estos métodos pueden producir una excepción de tipo **IllegalArgumentException** si *inicio* es mayor que *fin*, o una excepción **ArrayIndexOutOfBoundsException** si inicio o fin están fuera de límites. Las dos últimas formas pueden también producir una **ClassCastException** si los elementos del arreglo que se ordena no son comparables.

La clase **Arrays** además sobrescribe el método **toString( )** y **hashCode( )** para varios tipos de arreglos. Adicionalmente **deepToString( )** y **deepHashCode( )** trabajan sobre arreglos que contienen arreglos anidados.

El siguiente programa ilustra cómo utilizar algunos de los métodos de la clase **Arrays**:

```
// Ejemplo con Arrays
import java.util.*;

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

    // asignar memoria e inicializar arreglo
    int arreglo[] = new int[10];
    for(int i = 0; i < 10; i++)
      arreglo[i] = -3 * i;

    // mostrar, ordenar y mostrar nuevamente
    System.out.print("Contenidos originales: ");
    display(arreglo);
    Arrays.sort(arreglo);
    System.out.print("Ordenados: ");
    display(arreglo);

    // llenar y mostrar
    Arrays.fill(arreglo, 2, 6, -1);
    System.out.print("Tras llenar con fill(): ");
    display (arreglo);

    // ordenar y mostrar
    Arrays.sort (arreglo) ;
    System.out.print("Tras volver a ordenar: ");
    display(arreglo);

    // búsqueda binaria de -9
    System.out.print ("El valor -9 está en la posición ");
    int index =
      Arrays.binarySearch(arreglo, -9);

    System.out.println (index);
  }

  static void display(int arreglo[]) {
    for(int i : arreglo)
      System.out.print(i + " ");
```

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

Lo que sigue es la salida de este programa:

```
Contenidos originales: 0 -3 -6 -9 -12 -15 -18 -21 -24 -27
Ordenados: -27 -24 -21 -18 -15 -12 -9 -6 -3 0
Tras llenar con fill(): -27 -24 -1 -1 -1 -1 -9 -6 -3 0
Tras volver a ordenar: -27 -24 -9 -6 -3 -1 -1 -1 -1 0
El valor -9 está en la posición 2
```

¿Por qué colecciones con tipos parametrizados?

Como se mencionó al inicio de este capítulo, la estructura de colecciones completa fue equipada para trabajar con tipos parametrizados cuando JDK 5 fue liberado. Además, la estructura de colecciones es posiblemente uno de los más importantes usos de tipos parametrizados en el API de Java. La razón para esto es que los tipos parametrizados añaden seguridad al manejo de tipos en la estructura de colecciones. Antes de continuar, vale la pena tomar un poco de tiempo para examinar a detalle la importancia de esta mejora.

Comencemos con un ejemplo que utiliza código sin tipos parametrizados. El siguiente programa almacena una lista de cadenas en un **ArrayList** y luego despliega el contenido de la lista:

```
// Ejemplo de código sin tipos parametrizados
import java.util.*;

class OldStyle {
  public static void main(String args[]) {
    ArrayList lista = new ArrayList();

    // Estas líneas almacenan cadenas, pero cualquier tipo de objetos pueden
    // ser almacenados. En el código sin tipos parametrizados, no existe
    // una forma cómoda para restringir el tipo de objetos a almacenar
    // en la colección.
    lista.add("uno");
    lista.add("dos");
    lista.add("tres");
    lista.add("cuatro");

    Iterator itr = lista.iterator();
    while (itr.hasNext()) {

      // Para recuperar un elemento, se requiere una conversión explicita de tipos
      // debido a que la colección almacena elementos de tipo Object.
      String str = (String) itr.next(); // una conversión explícita de tipos se
requiere aquí.

      System.out.println(str + " tiene " + str.length() + "caracteres de
largo.");
    }
  }
}
```

Antes de los tipos parametrizados, todas las colecciones almacenaban referencias a tipos **Object**. Esto permitía que cualquier tipo de referencia fuera almacenada en la colección. El programa anterior utiliza esta característica para almacenar referencias a objetos de tipo **String** en la **lista**, pero cualquier tipo de eferencia podría haber sido almacenado.

Desafortunadamente, el hecho de que una colección sin tipos parametrizados almacenara referencias a **Object** podría fácilmente producir errores. Primero, es necesario que el programador y no el compilador, se asegure de que sólo objetos del tipo apropiado sean almacenados en una colección específica. Por ejemplo, en el ejemplo anterior, **lista** debe almacenar objetos de tipo **String**, pero no hay nada que realmente prevenga que otro tipo de referencia sea añadido a la colección. Por ejemplo, el compilador no encontrará ningún problema en esta línea de código:

```
lista.add(new Integer (100));
```

Debido a que **lista** almacena referencias de tipo **Object**, ésta puede almacenar una referencia a **Integer** de la misma forma que puede almacenar una referencia a **String**. Sin embargo, si se intenta mantener la lista sólo con referencias a **String**, la sentencia anterior corrompe la colección. De nuevo, el compilador no tiene forma de conocer que la sentencia anterior no es válida.

El segundo problema con las colecciones sin tipos parametrizados es que cuando se recupera una referencia de la colección, el programador debe manualmente convertir el tipo de la referencia al tipo adecuado. Por ello el programa anterior convierte la referencia entregada por **next( )** en un tipo **String**. Antes de los tipos parametrizados las colecciones simplemente almacenaban referencias a **Objetos**. Así, la conversión de tipos es necesaria para recuperar objetos de una colección.

Aparte de los inconvenientes de tener que convertir el tipo de cada referencia recuperada de una colección en el tipo adecuado, la falta de seguridad en el manejo de tipos a menudo conduce a un error serio y sorprendentemente fácil de producir. Debido a que **Object** puede ser convertido en cualquier tipo de objeto, es posible convertir una referencia obtenida de la colección en el *tipo equivocado*. Por ejemplo, si la siguiente sentencia fuera añadida en el ejemplo anterior, éste aún compilaría sin error, pero generaría una excepción en tiempo de ejecución con:

```
Integer i = (Integer) itr.next();
```

Recuerde que el ejemplo anterior almacena sólo referencias a instancias de tipo **String** en la **lista**. Así, cuando esta sentencia intenta convertir **String** a **Integer**, se genera una excepción que indica un error en la conversión. Debido a que esto ocurre en tiempo de ejecución, éste es un error muy serio.

La adición de tipos parametrizados mejora fundamentalmente la usabilidad y seguridad de las colecciones porque:

- Garantiza que sólo referencias a objetos del tipo adecuado pueden realmente ser almacenados en una colección. Así, una colección contendrá siempre referencias de tipo conocido.
- Elimina la necesidad de convertir una referencia recuperada de una colección. En lugar de ello, una referencia recuperada de una colección es convertida automáticamente en el tipo adecuado. Esto previene errores en tiempo de ejecución causados por conversiones erróneas de tipo.

Estas dos mejoras son posibles debido a que a cada clase de colección se le ha dado un parámetro de tipo que especifica el tipo de la colección. Por ejemplo, **ArrayList** ahora se declara como esto:

class ArrayList<E>

Donde, **E** es el tipo de los elementos almacenados en la colección. Por consiguiente, la siguiente línea declara un **ArrayList** para objetos de tipo **String**:

```
ArrayList<String> lista = new ArrayList<String>();
```

Ahora, sólo referencias de tipo **String** pueden ser añadidas a la **lista**.

Las interfaces **Iterator** y **ListIterator** también trabajan ahora con tipos parametrizados. Esto significa que el parámetro tipo debe coincidir con el tipo de la colección para la cual el iterador se ha obtenido. Además, esta compatibilidad de tipos es forzada en el momento de la compilación.

El siguiente programa muestra la forma moderna de escribir el programa anterior utilizando tipos parametrizados:

```
// Ejemplo de código con tipos parametrizados
import java.util.*;

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

    // Ahora la lista contiene referencias de tipo String
    ArrayList<String> lista = new ArrayList<String>();

    lista.add("uno");
    lista.add("dos");
    lista.add("tres");
    lista.add("cuatro");

    // Observe que el iterador utiliza también tipos parametrizados
    Iterator<String> itr = lista.iterator();

    // la siguiente sentencia causaría un error de compilación
Iterator<Integer> itr = lista.iterator(); // error

    while (itr.hasNext()) {
      String str = itr.next(); // no se requiere una conversión explícita de tipos

      // Ahora bien, la siguiente línea causa un error en tiempo de compilación
      // ya no en tiempo de ejecución.
// Integer i = itr.next(); // esto no compila

      System.out.println(str + " tiene " + str.length() + "caracteres de
        largo.");
    }
  }
}
```

Ahora, **lista** sólo puede contener referencias a objetos de tipo **String**. Además, la siguiente línea muestra que no es necesario convertir el valor devuelto por **next( )** en un objeto **String**:

```
String str = itr.next(); // no es necesaria ninguna conversión de tipos.
```

La conversión de tipos es realizada automáticamente.

Debido al soporte brindado a los tipos en bruto, no es necesario actualizar inmediatamente programas antiguos que utilizaban colecciones. Sin embargo, todos los nuevos programas deben emplear tipos parametrizados, y debemos actualizar los programas anteriores tan pronto el tiempo lo permita. La adición de los tipos parametrizados a la estructura de colecciones es una mejora fundamental que debe ser utilizada siempre que sea posible.

Las clases e interfaces preexistentes

Como se explicó al inicio de este capítulo, la versión original de **java.util** no incluía a la estructura de colecciones. En su lugar, definía varias clases y una interfaz que proporcionaba un método *ad hoc* para almacenar objetos. Con la incorporación de las colecciones en J2SE 1.2, varias de las clases originales fueron reconstruidas para soportar las interfaces de colección. Por ello, son totalmente compatibles con la estructura de colecciones. Aunque no se han desechado clases, algunas se dan por obsoletas. Por supuesto, cuando una colección duplica la funcionalidad de una clase preexistente, lo normal es que se quiera usar la colección para escribir nuevo código.

Un nota adicional: ninguna de las clases de colección están sincronizadas, pero todas las clases preexistentes están sincronizadas. Esta distinción puede ser importante en algunas situaciones. Por supuesto, también se puede sincronizar colecciones fácilmente, utilizando alguno de los algoritmos proporcionados por **Collections**.

Las clases preexistentes definidas por **java.util** se muestran aquí:

Dictionary	Hashtable	Properties	Stack	Vector

Hay una interfaz preexistente llamada **Enumeration**. Las siguientes secciones analizan **Enumeration** y cada una de las clases preexistentes.

La interfaz Enumeration

La interfaz **Enumeration** define los métodos con los cuales se puede *enumerar* (obtener de uno en uno) los elementos de una colección de objetos. Esta interfaz preexistente ha sido remplazada por **Iterator**. Aunque no se ha desechado, **Enumeration** se considera obsoleta para escribir código nuevo. Sin embargo, es utilizada por varios métodos definidos por las clases preexistentes (como **Vector** y **Properties**), es utilizada por muchas otras clases del API, y su uso está actualmente muy extendido. Debido a que aún está en uso, la interfaz fue mejorada con tipos parametrizados en el JDK 5. La interfaz está declarada como:

```
interface Enumeration<E>
```

Donde **E** especifica el tipo de los elementos que son enumerados.

Enumeration especifica los siguientes dos métodos:

```
boolean hasMoreElements( )
E nextElement( )
```

Al implementarse, **hasMoreElements( )** debe devolver **true** mientras haya más elementos que extraer, y **false** cuando todos los elementos se han enumerado. **nextElement( )** devuelve el siguiente objeto en la enumeración. Esto es, cada llamada a **nextElement( )** obtiene el siguiente objeto en la enumeración. Este método genera una excepción de tipo **NoSuchElementException** cuando la enumeración está completa.

Vector

Vector implementa un arreglo dinámico. Es similar a **ArrayList**, pero con dos diferencias: **Vector** está sincronizado, y contiene muchos métodos preexistentes que no son parte de la estructura de colecciones.

Con la llegada de la estructura de colecciones, **Vector** fue rehecho para extender **AbstractList** e implementar la interfaz **List**. Con la llegada del JDK 5, **Vector** fue mejorado para soportar tipos parametrizados y rehecho para implementar a **Iterable**. Hoy día **Vector** es totalmente compatible con las colecciones, y **Vector** permite que su contenido sea recorrido utilizando un ciclo **for-each**.

La clase **Vector** está declarada como:

```
class Vector<E>
```

Donde, **E** especifica el tipo de elementos que serán almacenados.

Aquí están los constructores **Vector**:

```
Vector( )
Vector(int tamaño)
Vector(int tamaño, int i)
Vector(Collection<? extends E> c)
```

La primera forma crea un vector por omisión, que tiene un tamaño inicial de 10. La segunda forma crea un vector cuya capacidad viene especificada por *tamaño*. La tercera forma crea un vector de capacidad especificada en *tamaño* y cuyo incremento está especificado por *i*. El incremento indica el número de elementos cuya memoria se asigna cada vez que el vector se aumenta de tamaño. La cuarta forma crea un vector que contiene los elementos de la colección *c*.

Todos los vectores comienzan con una capacidad inicial. Tras ser completada la capacidad inicial, la siguiente vez que se intenta guardar un objeto en el vector, el vector automáticamente asigna espacio para ese objeto y espacio extra para objetos adicionales. Asignando más memoria que la requerida, el vector reduce el número de veces que hay que hacer asignaciones de memoria. Esta reducción es importante, porque las reasignaciones son costosas en términos de tiempo. La cantidad de espacio extra asignado durante cada reasignación se determina por el incremento especificado al crear el vector. Si no se especifica un incremento, el tamaño del vector se duplica.

Vector define los siguientes datos miembros protegidos:

```
int capacityIncrement;
int elementCount;
Object[ ] elementData;
```

El valor del incremento se almacena en **capacityIncrement**. El número de elementos actualmente en el vector se almacena en **elementCount**. El arreglo que contiene el vector se almacena en **elementData**.

Además de los métodos de colecciones definidos por List, Vector incluye varios métodos preexistentes, que se muestran en la Tabla 17-15.

Dado que **Vector** implementa **List**, se puede utilizar un vector igual que se utiliza una instancia de **ArrayList**. También se puede manipular una utilizando sus métodos preexistentes. Por ejemplo, tras crear una instancia de Vector, se le puede añadir un elemento llamando a **addElement( )**. Para obtener el elemento en una posición específica basta con llamar a **elementAt( )**. Para obtener el primer elemento en el Vector, se ha de llamar a **firstElement( ).** Para recuperar el último elemento, se llama a **lastElement( )**. Se puede obtener el índice de un elemento utilizando **indexOf( )** y **lastIndexOf( )**. Para quitar un elemento se llama a **removeElement( )** o **removeElementAt( )**.

Método	Descripción
void addElement(E *elemento*)	El objeto especificado por *elemento* se añade al vector.
int capacity( )	Devuelve la capacidad del vector.
Object clone( )	Devuelve un duplicado del vector que invoca.
boolean contains(Object *elemento*)	Devuelve **true** si *elemento* está contenido en el vector, y devuelve **false** en caso contrario.
void copyInto(Object *a* [ ])	Los elementos contenidos en el vector que invoca se copian en el arreglo especificado por *a*.
E elementAt(int *i*)	Devuelve el elemento en la posición indicada por *i*.
Enumeration<E> elements( )	Devuelve una enumeración de los elementos en el vector.
void ensureCapacity(int *tamaño*)	Establece la mínima capacidad del vector con el valor de *tamaño*.
E firstElement( )	Devuelve el primer elemento del vector.
int indexOf(Object *elemento*)	Devuelve el índice de la primera aparición de *elemento*. Si el objeto no está en el vector devuelve –1.
int indexOf(Object *elemento*, int *inicio*)	Devuelve el índice de la primera aparición de *elemento* en o después de *inicio*. Si el objeto no está en esa parte del vector, se devuelve –1.
void insertElementAt(E *elemento*, int *indice*)	Añade *elemento* al vector en la posición indicada por *indice*.
boolean isEmpty( )	Devuelve **true** si el vector está vacío y **false** si contiene uno o más elementos.
E lastElement( )	Devuelve el último elemento del vector.
int lastIndexOf(Object *elemento*)	Devuelve el índice de la última aparición de *elemento*. Si el objeto no está en el vector, se devuelve –1.
int lastIndexOf(Object *elemento*, int *inicio*)	Devuelve el índice de la última aparición de *elemento* antes de *inicio*. Si el objeto no está en esa parte del vector, se devuelve –1.
void removeAllElements( )	Vacía el vector. Tras ejecutarse este método, el tamaño del vector es cero.
boolean removeElement(Object *elemento*)	Quita *elemento* del vector. Si existe en el vector más de una instancia del objeto especificado, entonces se quita la primera. Devuelve **true** si se hizo con éxito y **false** si el objeto no se encontró.
void removeElementAt(int *indice*)	Quita el elemento en la posición indicada por *indice*.
void setElementAt(E *elemento*, int *indice*)	Se asigna *elemento* a la posición indicada por *indice*.
void setSize(int *tamaño*)	Establece el número de elementos en el vector con el valor de *tamaño*. Si el nuevo tamaño es menor que el viejo, se pierden elementos. Si el nuevo tamaño es mayor que el viejo, se añaden elementos con valor **null**.
int size( )	Devuelve el número de elementos en el vector.
String toString( )	Devuelve la cadena equivalente del vector.
void trimToSize( )	Establece la capacidad del vector para que sea igual al número de elementos que contiene actualmente.

TABLA 17-15 Los métodos definidos por **Vector**

El siguiente programa utiliza un vector para almacenar varios tipos de objetos numéricos. Ejemplifica el uso de varios métodos preexistentes definidos por **Vector**. También ilustra la interfaz **Enumeration**.

```
// Ejemplo de operaciones con Vector.
import java.util.*;

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

    // el tamaño inicial es 3, el incremento es 2

    Vector<Integer> v = new Vector<Integer>(3, 2);

    System.out.println ("Tamaño inicial: " + v. size());
    System.out.println("Capacidad inicial: " +
                       v.capacity());

    v.addElement(l);
    v.addElement(2);
    v.addElement(3);
    v.addElement(4);

    System.out.println("Capacidad tras cuatro adiciones: " +
                        v.capacity());
    v.addElement(5);
    System.out.println("Capacidad actual: " +
                       v.capacity());

    v.addElement(6);
    v.addElement(7);

    System.out.println("Capacidad actual: " +
                       v.capacity());

    v.addElement(9);
    v.addElement(10);

    System.out.println("Capacidad actual: " +
                       v.capacity());

    v.addElement(11);
    v.addElement(12);


    System.out.println("Primer elemento: " + v.firstElement());
    System.out.println("Último elemento: " + v.lastElement());

    if(v.contains(3))
      System.out.println("El Vector contiene el número 3.");

    // enumerar los elementos del Vector.
    Enumeration vEnum = v.elements();

    System.out.println("\nElementos del Vector:");
    while(vEnum.hasMoreElements())
      System.out.print(vEnum.nextElement() + " ");
    System.out.println();
  }
}
```

La salida de este programa se muestra a continuación:

```
Tamaño inicial: 0
Capacidad inicial: 3
Capacidad tras cuatro adiciones: 5
```

```
Capacidad actual: 5
Capacidad actual: 7
Capacidad actual: 9

Primer elemento: 1
Último elemento: 12
El Vector contiene el número 3.

Elementos del Vector:
1 2 3 4 5 6 7 9 10 11 12
```

En vez de basarse en una enumeración para llevar a cabo un ciclo a lo largo de los objetos (como lo hace el programa anterior), ahora se puede utilizar un iterador. Por ejemplo, se puede sustituir en el programa el siguiente código que utiliza iteradores:

```
// utilizar un iterador para mostrar los contenidos
Iterator<Integer> vItr = v.iterator();

System.out.println("\nElementos del Vector:");
while(vItr.hasNext())
  System.out.print (vItr.next () + " ");
System.out.println();
```

También se puede utilizar un ciclo **for-each** para recorrer el **Vector**, como se muestra en la siguiente versión del código anterior:

```
// utilizar un ciclo for-each para mostrar los contenidos
System.out.println("\nElementos del Vector:");
For (int i : v)
  System.out.print (i + " ");

System.out.println();
```

Debido a que el uso de la interfaz **Enumeration** no se recomienda para nuevos programas, habitualmente se utilizará un iterador o un ciclo **for-each** para enumerar los contenidos de un vector. Por supuesto, existe mucho código preexistente que emplea la **enumeración**. Afortunadamente, las iteraciones y las enumeraciones funcionan casi de la misma manera.

Stack

Stack es una subclase de **Vector** que implementa una pila estándar bajo el modelo último en entrar-primero en salir. **Stack** sólo define el constructor por omisión, que crea una pila vacía. Con JDK 5, la clase **Stack** fue mejorada con tipos parametrizados. La clase está declarada como:

class Stack<E>

Donde, **E** especifica el tipo de elementos almacenados en la pila.

Stack incluye todos los métodos definidos por **Vector** y añade varios métodos propios, mostrados en la Tabla 17-16.

Para poner un objeto encima de la pila, se llama al método **push( )**. Para quitar y devolver el elemento en la posición más alta, se llama al método **pop( )**. Se produce una excepción **EmptyStackException** si se llama a **pop( )** y la pila que invoca está vacía. Se puede utilizar el método **peek( )** para devolver, pero no quitar, el objeto en la posición más alta. El método **empty( )** devuelve **true** si no hay nada en la pila. El método **search( )** determina si un objeto existe en la pila, y devuelve el número de veces que hay que hacer **pop( )** para llevarlo a la posición más alta de la pila.

Método	Descripción
boolean empty( )	Devuelve **true** si la pila está vacía, y devuelve **false** si la pila contiene elementos.
E peek( )	Devuelve el elemento en lo alto de la pila sin removerlo.
E pop( )	Devuelve y remueve el elemento en lo alto de la pila.
Object push(E *elemento*)	Introduce *elemento* en la pila. También devuelve *elemento*.
int search(Object *elemento*)	Busca *elemento* en la pila. Si lo encuentra, se devuelve su desplazamiento desde lo alto de la pila. De lo contrario, se devuelve –1.

TABLA 17-16 Los métodos definidos por **Stack**

Aquí hay un ejemplo que crea una pila, introduce varios objetos **Integer** en ella, y luego los saca de nuevo:

```
// Ejemplo con la clase Stack
import java.util.*;

class StackDemo {
  static void showpush(Stack<Integer> st, int a) {
    st.push(a);
    System.out.println("push(" + a + ")");
    System.out.println("pila: " + st);
  }

  static void showpop(Stack<Integer> st) {
    System.out.print("pop -> ");
    Integer a = st.pop();
    System.out.println(a);
    System.out.println("pila: " + st);
  }

  public static void main(String args[]) {
    Stack<Integer> st = new Stack<Integer>();

    System.out.println("pila: " + st);
    showpush(st, 42);
    showpush(st, 66);
    showpush(st, 99);
    showpop(st);
    showpop(st);
    showpop(st);

    try{
      showpop(st);
    } catch (EmptyStackException e) {
      System.out.println("pila vacía");
    }
  }
}
```

La siguiente es la salida producida por el programa; nótese cómo se utiliza el gestor de excepciones para gestionar a **EmptyStackException** elegantemente en el caso de la pila vacía:

```
pila: [ ]
push(42)
pila: [42]
```

```
push(66)
pila: [42, 66]
push(99)
pila: [42, 66, 99]
pop -> 99
pila: [42, 66]
pop -> 66
pila: [42]
pop -> 42
pila: [ ]
pop -> pila vacía
```

Finalmente, aunque el uso de **Stack** no está descontinuado, con la liberación de Java SE 6, **ArrayDequeue** es una mejor opción.

Dictionary

Dictionary es una clase abstracta que representa un almacén de pares clave/valor y opera en gran medida como **Map**. Dados una clave y un valor, se puede almacenar el valor en el objeto **Dictionary**. Una vez que el valor se ha almacenado, se puede recuperar utilizando su clave. Así, al igual que un mapa, un diccionario se puede entender como una lista de pares clave/valor. Aunque no se ha desechado su uso, **Dictionary** se ha clasificado como obsoleto, porque ha sido superado por **Map**. Sin embargo, **Dictionary** es en la actualidad ampliamente utilizado, por lo que lo trataremos en su totalidad.

Con JDK 5, se agregó a **Dictionary** el uso de tipos parametrizados. La clase está declarada como sigue:

class Dictionary<K,V>

Donde, **K** especifica el tipo de las claves y **V** especifica el tipo de los valores. Los métodos abstractos definidos en **Dictionary** se enlistan en la Tabla 17-17.

Método	Descripción
Enumeration<V> elements( )	Devuelve la enumeración de los valores contenidos en el diccionario.
V get(Object *k*)	Devuelve el objeto que contiene el valor asociado con *k*. Si *k* no está en el diccionario, se devuelve un objeto **null**.
boolean isEmpty( )	Devuelve **true** si el diccionario está vacío, y devuelve **false** si contiene al menos una clave.
Enumeration<K> keys( )	Devuelve una enumeración de las claves contenidas en el diccionario.
V put(K *clave*, V *valor*)	Inserta una clave y su valor en el diccionario. Devuelve **null** si clave no está ya en el diccionario; devuelve el valor previo asociado a clave si clave ya está en el diccionario.
V remove(Object *clave*)	Quita *clave* y su valor. Devuelve el valor asociado a *clave*. Si *clave* no está en el diccionario, se devuelve **null**.
int size( )	Devuelve el número de entradas en el diccionario.

TABLA 17-17 Los métodos abstractos definidos por **Dictionary**

Para añadir una clave y un valor se utiliza el método **put( ).** Se utiliza **get( )** para recuperar el valor de una clave dada. Tanto las claves como los valores pueden ser devueltos como una enumeración por los métodos **keys( )** y **elements( ),** respectivamente. El método **size( )**

devuelve el número de pares clave/valor almacenados en un diccionario, y **isEmpty( )** devuelve **true** cuando el diccionario está vacío. Se puede utilizar el método **remove( )** para borrar un par clave/valor.

NOTA *La clase **Dictionary** es obsoleta. Es mejor implementar la interfaz Map para obtener la funcionalidad del almacenamiento de pares clave/valor.*

Hashtable

Hashtable fue parte del paquete original **java.util**, y es una implementación concreta de **Dictionary**. Sin embargo, con la llegada de la estructura de colecciones, la clase **Hashtable** fue reconstruida para que implementara la interfaz **Map**. Así, **Hashtable** está ahora integrada en la estructura de colecciones. Es similar a **HashMap**, pero está sincronizada.

Al igual que **HashMap**, **Hashtable** almacena pares clave/valor en una tabla de dispersión. Sin embargo ni las claves ni los valores pueden ser **null**. Cuando se utiliza **Hashtable** se especifica un objeto que se usa como clave y el valor que se desea vincular a esa clave. La clave entonces se dispersa, y el código de dispersión resultante se utiliza como el índice con el que el valor se almacena dentro de la tabla.

Hashtable utiliza tipos parametrizados a partir del JDK 5 y está declarada como:

```
class Hashtable<K,V>
```

Donde, **K** especifica el tipo de las claves, y **V** especifica el tipo de los valores.

Una tabla de dispersión sólo puede almacenar objetos que sobrescriban los métodos **hashCode( )** y **equals(**) definidos por **Object**. El método **hashCode( )** debe calcular y devolver el código de dispersión del objeto. Por supuesto, **equals( )** compara dos objetos. Afortunadamente, muchas de las clases incorporadas en Java ya implementan el método **hashCode( ).** Por ejemplo, el tipo más común de **Hashtable** usa un objeto **String** como clave. **String** implementa tanto **hashCode( )** como **equals( ).**

Los constructores de **Hashtable** son:

```
Hashtable( )
Hashtable(int tamaño)
Hashtable(int tamaño, float razonLlenado)
Hashtable(Map<? extends K, ? extends V> m)
```

La primera versión es el constructor por omisión. La segunda versión crea una tabla de dispersión con un tamaño inicial indicado en *tamaño* (el tamaño por omisión es 11). La tercera versión crea una tabla de dispersión con un *tamaño* inicial indicado en *tamaño* y una razón de llenado indicada por *razonLlenado*. Este valor debe estar comprendido entre 0.0 y 1.0, y determina lo llena que puede estar la tabla de dispersión antes de aumentarse su tamaño. Específicamente, cuando el número de elementos es mayor que la capacidad de la tabla de dispersión multiplicada por su razón de llenado, la tabla de dispersión se expande. Si no se especifica una razón de llenado, se utiliza 0.75. Por último, la cuarta versión crea una tabla de dispersión que se inicializa con los elementos de *m*. La capacidad de la tabla de dispersión se fija en el doble del número de elementos de *m*. Se utiliza el factor de carga por omisión de 0.75.

Además de los métodos definidos por la interfaz **Map**, que **Hashtable** implementa ahora, **Hashtable** define los métodos preexistentes listados en la Tabla 17-18. Muchos métodos generan la excepción **NullPointerException** si se intenta insertar un elemento **null** como clave o valor.

Método	Descripción
void clear( )	Reinicia y vacía la tabla de dispersión.
Object clone( )	Devuelve un duplicado del objeto que invoca.
boolean contains(Object *valor*)	Devuelve **true** si existe en la tabla de dispersión algún valor igual a *valor*. Devuelve **false** si el valor no se encuentra.
boolean containsKey(Object *clave*)	Devuelve **true** si en la tabla de dispersión existe alguna clave igual a *clave*. Devuelve **false** si la clave no se encuentra.
boolean containsValue(Object *valor*)	Devuelve **true** si existe algún valor igual a *valor* dentro de la tabla de dispersión. Devuelve **false** si el valor no se encuentra.
Enumeration<V> elements( )	Devuelve una enumeración de los valores contenidos en la tabla de dispersión.
V get(Object clave)	Devuelve el objeto que contiene el valor asociado con clave. Si clave no está en la tabla de dispersión, se devuelve un objeto **null**.
boolean isEmpty( )	Devuelve **true** si la tabla de dispersión está vacía; devuelve **false** si contiene al menos una clave.
Enumeration<K> keys( )	Devuelve una enumeración de las claves contenidas en la tabla de dispersión.
V put(K *clave*, V *valor*)	Inserta una clave y un valor en la tabla de dispersión. Devuelve **null** si *clave* no está ya en la tabla de dispersión; devuelve el *valor* anteriormente asociado con *clave* si clave está ya en la tabla de dispersión.
void rehash( )	Aumenta el tamaño de la tabla de dispersión y redispersiona todas sus claves.
V remove(Object *clave*)	Elimina a *clave* y su valor asociado de la tabla. Devuelve el valor asociado a *clave*. Si *clave* no está en la tabla de dispersión, se devuelve un objeto **null**.
int size( )	Devuelve el número de entradas en la tabla de dispersión.
String toString( )	Devuelve la cadena equivalente de una tabla de dispersión.

TABLA 17-18 Los métodos preexistentes definidos por **Hashtable**

El siguiente ejemplo rehace el programa de las cuentas bancarias, antes mostrado, de modo que utilice una **Hashtable** para almacenar los nombres de los titulares de las cuentas y sus saldos actualizados:

```
// Ejemplo de Hashtable
import java.util.*;

class HTDemo {
  public static void main(String args[]) {
    Hashtable<String, Double> balance =
      new Hashtable<String, Double> ();

    Enumeration<String> names;
    String str;
    double bal;

    balance.put("Ken Bauer", 3434.34);
    balance.put("Tom Smith", 123.22);
    balance.put("Jane Baker", 1378.00);
    balance.put("Todd Hall", 99.22);
    balance.put("Ralph Smith", -19.08);
```

```
    // Mostrar todos los saldos en la tabla de dispersión.
    names = balance.keys () ;

    while(names.hasMoreElements()) {
      str = names.nextElement();
      System.out.println(str + ": " + balance.get(str));
    }

    System.out.println();

    // Ingresar 1,000 en la cuenta de Ken Bauer
    bal = balance.get("Ken Bauer");
    balance.put("Ken Bauer", bal+1000);
    System.out.println("Nuevo saldo de Ken Bauer: " +
                       balance.get("Ken Bauer"));
  }
}
```

La salida del programa se muestra aquí:

```
Todd Hall: 99.22
Ralph Smith: -19.08
Ken Bauer: 3434.34
Jane Baker: 1378.0
Tom Smith: 123.22

Nuevo saldo de Ken Bauer: 4434.34
```

Cabe destacar que, al igual que las clases para mapas, **Hashtable** no soporta directamente iteradores. Por tanto, el programa anterior utiliza una enumeración para mostrar los contenidos de **balance**. Sin embargo, es posible obtener vistas de conjunto de la tabla de dispersión, lo que permite el uso de iteradores. Para hacer esto, basta con usar uno de los métodos de vista de colección definidos por **Map**, como **entrySet( )** o **keySet( )**. Por ejemplo, se puede conseguir una vista de conjunto de las claves e iterar a lo largo de ellas utilizando ya sea un iterador o un ciclo **for-each**. Aquí tenemos una versión rehecha del programa anterior para ilustrar esta técnica:

```
// Uso de iteradores con una tabla de dispersión.
import java.util.*;

class HTDemo2 {
  public static void main(String args[]) {
    Hashtable<String, Double> balance =
      new Hashtable<String, Double> ();

    String str;
    double bal;

    balance.put("Ken Bauer", 3434.34);
    balance.put("Tom Smith", 123.22);
    balance.put("Jane Baker", 1378.00);
    balance.put("Todd Hall", 99.22);
    balance.put("Ralph Smith", -19.08);

    // Mostrar todos los saldos en la tabla de dispersión
    // Primero, se obtiene una vista en conjunto de las claves

    Set<String> set = balance.keySet();

    // obtener iterador
    Iterator<String> itr = set.iterator();
    while(itr.hasNext()) {
```

```
      str = itr.next();
      System.out.println(str + ": " +
                         balance.get(str));
    }
    System.out.println();
    // Ingresar 1,000 en la cuenta de Ken Bauer
    bal = balance.get("Ken Bauer");
    balance.put ("Ken Bauer", bal+1000);
    System.out.println ("Nuevo saldo de Ken Bauer: " +
                        balance.get("Ken Bauer"));
  }
}
```

Properties

Properties es una subclase de **Hashtable**. Se utiliza para mantener una lista de valores en los que la clave y el valor son **String**. La clase **Properties** es utilizada por muchas otras clases de Java. Por ejemplo, es el tipo de objeto devuelto por **System.getProperties( )** al obtener valores de entorno. Aunque la clase **propierties**, por sí misma, no utiliza tipos parametrizados, muchos de sus métodos si lo hacen.

Properties define la siguiente variable de instancia:

Properties defaults;

Esta variable contiene una Lista de propiedades por omisión asociada a un objeto **Properties**. **Properties** define estos constructores:

Properties( )
Properties(Properties *prop*)

La primera versión crea un objeto **Properties** sin valores por omisión. La segunda crea un objeto que toma sus valores por omisión del parámetro *prop*. En ambos casos la lista de propiedades está vacía.

Además de los métodos que **Properties** hereda de **Hashtable**, **Properties** define los métodos listados en la Tabla 17-19. **Properties** también contiene un método desechado: **save( )**. Éste fue remplazado por **store( )** porque **save( )** no manejaba adecuadamente los errores.

Una posibilidad útil que aporta la clase **Properties** es que se puede especificar una propiedad por omisión a devolver si no hay ningún valor asociado con determinada clave. Por ejemplo, se puede especificar un valor por omisión junto con la clave en el método **getProperty( )**, –como **getProperty("nombre", "valor por omisión").** Si el valor "nombre"no se encuentra, entonces se devuelve "valor por omisión". Al construir un objeto **Properties**, se puede pasar otra instancia de **Properties** que proporcione las propiedades por omisión para la nueva instancia. En este caso, si se llama a **getProperty("algo")** sobre un objeto **Properties** dado, y "algo"no existe, Java busca "algo"en el objeto **Properties** por omisión. Esto permite la anidación arbitraria de niveles de propiedades por omisión.

Método	Descripción
String getProperty(String *clave*)	Devuelve el valor asociado a *clave*. Si *clave* no está ni en la lista ni en la lista de propiedades por omisión, se devuelve un objeto **null**.

TABLA 17-19 Los métodos preexistentes definidos por **Properties**

Método	Descripción
String getProperty(String *clave*, String *prop*)	Devuelve el valor asociado a *clave*. Se devuelve *prop* si *clave* no está en la lista ni en la lista de propiedades por omisión.
void List(PrintStream *flujoSal*)	Envía la lista de propiedades al flujo de salida vinculado a *flujoSal*.
void List(PrintWriter *flujoSal*)	Envía la lista de propiedades al flujo de salida vinculado a *flujoSal*.
void load(InputStream *flujoEnt*) throws IOException	Recibe una Lista de propiedades del flujo de entrada vinculado a *flujoEnt*.
void load(Reader *flujoEnt*) throws IOException	Recibe una lista de propiedades del flujo de entrada vinculado a *flujoEnt* (Añadido por Java SE 6).
void loadFromXML(InputStream *flujoEnt*) throws IOException, InvalidPropertiesFormatException	Recibe una lista de propiedades desde un documento XML vinculado a *flujoEnt*.
Enumeration<?> propertyNames( )	Devuelve una enumeración de las claves. Ésta también incluye las claves encontradas en la Lista de propiedades por omisión.
Object setProperty(String *clave*, String *valor*)	Asocia *valor* con *clave*. Devuelve el valor previamente asociado con *clave*, o devuelve **null** si no existía tal asociación.
void store(OutputStream *flujoSal*, String *desc*) throws IOException	Después de escribir la cadena especificada por *desc*, la Lista de propiedades se escribe al flujo de salida vinculado a *flujoSal*.
void store(Writer *flujoSal*, String *desc*) throws IOException	Después de escribir la cadena especificada por *desc*, la Lista de propiedades se escribe al flujo de salida vinculado a *flujoSal*. Añadido por Java SE 6.
void storeToXML(OutputStream *flujoSal*, String *desc*) throws IOException	Después de escribir la cadena especificada por *desc*, la Lista de propiedades se escribe al documento XML vinculado a *flujoSal*.
void storeToXML(OutputStream *flujoSal*, String *desc*, String *enc*)	La Lista de propiedades y la cadena especificada por *desc* son escritas en el flujo de salida vinculado a *flujoSal* utilizando la codificación de caracteres especificada en *enc*.
Set<String> stringPropertyNames( )	Devuelve un conjunto de claves (añadido por Java SE 6).

TABLA 17-19 Los métodos preexistentes definidos por **Properties** *(continuación)*

El siguiente ejemplo muestra el uso de **Properties**. Crea una lista de propiedades en que las claves son los nombres de países, y los valores son los nombres de sus capitales. Nótese que el intento de encontrar la capital de Francia incluye un valor por omisión.

```
// Ejemplo para ilustrar una lista de propiedades.
import java.util.*;

class PropDemo {
  public static void main(String args[]) {
    Properties capitals = new Properties();

    capitals.put ("España", "Madrid");
    capitals.put("Argentina", "Buenos Aires");
    capitals.put("México", "Ciudad de México");
    capitals.put("El Salvador", "San Salvador");
    capitals.put("Colombia", "Bogotá");
```

```
    // Obtener una vista de conjunto para las claves
    Set paises = capitals.keySet();

    // Mostrar todos los países y capitales.
    for (Object name : paises)
      System.out.println("La capital de " +
                         name + " es " + capitals.getProperty((String)name)
                         + ".");

    System.out.println();

    // Busca un país que no está en la Lista y especifica un valor por omisión
    String str = capitals.getProperty("Francia", "No Encontrada");
    System.out.println("La capital de Francia es:
                       + str + ".");
  }
}
```

La salida de este programa se muestra aquí:

```
La capital de España es Madrid.
La capital de Argentina es Buenos Aires.
La capital de México es Ciudad de México.
La capital de El Salvador es San Salvador.
La capital de Colombia es Bogotá.

La capital de Francia es: No Encontrada.
```

Como Francia no está en la lista, se utiliza el valor por omisión.

Aunque es perfectamente válido utilizar un valor por omisión cuando se llama a **getProperty( )**, como muestra el ejemplo precedente, existe una mejor forma de gestionar los valores por omisión para la mayoría de las aplicaciones con las listas de propiedades. Para mayor flexibilidad, se especifica una lista de propiedades por omisión al construir un objeto **Properties**. En la lista por omisión se buscará si la clave deseada no se encuentra en la lista principal. Por ejemplo, la siguiente es una versión ligeramente modificada del programa anterior, que especifica una lista de países por omisión. Ahora, cuando se busque Francia, se encontrará en la lista por omisión:

```
// Uso de una Lista de propiedades por omisión.
import java.util.*;

class PropDemoDef {
  public static void main(String args[]) {
    Properties defList = new Properties();
    defList.put("Francia", "París");
    defList.put("Portugal", "Lisboa");

    Properties capitals = new Properties(defList);

    capitals.put("España", "Madrid");
    capitals.put("Argentina", "Buenos Aires");
    capitals.put("México", "Ciudad de México");
    capitals.put("El Salvador", "San Salvador");
    capitals.put("Colombia", "Bogotá");

    // Obtener una vista de conjunto para las claves
    Set paises = capitals.keySet();

    // Mostrar todos los países y capitales.
    for (Object name : paises)
```

```
      System.out.println("La capital de " + name +
                         " es " + capitals.getProperty((String)name)
                         + ".");

    System.out.println();

    // Francia será encontrada en la lista por omisión
    String str = capitals.getProperty("Francia");
    System.out.println("La capital de Francia es:
                       + str + ".");
  }
}
```

Uso de store( ) y load( )

Uno de los aspectos más útiles de **Properties** es que la información contenida en un objeto **Properties** se puede almacenar o leer de un disco fácilmente con los métodos **store( )** y **load( )**. En cualquier momento se puede escribir un objeto **Properties** en un canal de datos, o leerlo. Esto hace a las listas de propiedades especialmente cómodas para implementar bases de datos sencillas. Por ejemplo, el siguiente programa utiliza una lista de propiedades para crear una sencilla lista telefónica computarizada que almacena nombres y números de teléfono. Para encontrar el número de una persona, se introduce su nombre. El programa utiliza los métodos **store( )** y **load( )** para almacenar y recuperar la lista. Cuando el programa se ejecuta, primero intenta cargar la lista de un archivo llamado **Listatelef.dat.** Si este archivo existe, la lista se carga. Entonces se puede añadir datos a la lista. Si se hace, la nueva lista se salva al terminar el programa. Nótese el poco código que se requiere para implementar una lista telefónica computarizada sencilla pero funcional.

```
/*  Una sencilla base de datos con números telefónicos
    que utiliza una Lista de propiedades. */
import java.io.*;
import java.util.*;

class Phonebook {
  public static void main(String args[])
    throws IOException
  {
    Properties ht = new Properties();
    BufferedReader br = new BufferedReader(new InputStrearnReader(System.in));
    String nombre, numero;
    FileInputStream fin = null;
    boolean cambiado = false;

    // Intentar abrir el archivo Listatelef.dat
    try {
      fin = new FileInputStream ("Listatelef.dat");
    } catch(FileNotFoundException e) {
    // ignora el error de archivo inexistente
    }

    /*  Si el archivo Listatelef ya existe,
        cargar los números de teléfono existentes. */
    try {
      if (fin != null) {
        ht.load(fin);
        fin.close();
```

```
      }
    } catch(IOException e) {
      System.out.println("Error leyendo archivo.");
    }

    // dejar que el usuario introduzca nuevos nombres y números.
    do{
      System.out.println("Introducir nuevo nombre" +
                         " (escriba 'fin' para terminar): ");
      nombre = br.readLine();
      if(nombre.equals("fin")) continue;

      System.out.println("Introducir número: ");
      número = br.readLine();

      ht.put(nombre, número);
      cambiado = true;
    } while( !nombre.equals("fin"));

    // si la Lista telefónica ha cambiado, guardarla.
    if(cambiado) {
      FileOutputStream fout = new FileOutputStream ("Listatelef.dat");

      ht.store(fout, "Lista Telefónica");
      fout. close();
    }

    // buscar números dado un nombre.
    do {
      System.out.println("Introducir nombre a encontrar" +
                         " ('fin' para terminar): ");
      nombre = br.readLine();
      if(nombre.equals("fin")) continue;

      número = (String) ht.get(nombre);
      System.out.println(número);
    } while (!nombre.equals("fin"));
  }
}
```

Resumen de las Colecciones

La estructura de colecciones le da al programador un potente conjunto de soluciones bien construidas para algunas de las tareas de programación más comunes. Ahora que la estructura de colecciones es genérica y utiliza tipos parametrizados, ésta puede ser utilizada con una completa seguridad de tipos, lo cual además contribuye a darle relevancia. Considere el uso de una colección la próxima vez que necesite almacenar y recuperar información. Recuerde que las colecciones no se tienen que reservar exclusivamente para esos"trabajos grandes", como bases de datos corporativas, Listas de correo o sistemas de inventario. Pueden ser también efectivas cuando se aplican a trabajos más pequeños. Por ejemplo, un **TreeMap** sería una colección excelente para contener la estructura de directorios de un conjunto de archivos. Un **TreeSet** podría ser útil para almacenar información de gestión de proyectos. Francamente, los tipos de problemas que se beneficiarán de una solución basada en las colecciones sólo pueden encontrar un límite en nuestra imaginación.

CAPÍTULO 18

java.util parte 2: más clases de utilería

Este capítulo continúa nuestro análisis de **java.util** examinando las clases e interfaces que no forman parte de la estructura de colecciones. Éstas incluyen clases que separan cadenas en partes, trabajan con fechas, calculan números aleatorios, agrupan recursos y observan eventos. También se mencionan las clases **Formatter** y **Scanner**, las cuales facilitan la escritura y lectura de datos con formato. Al final de este capítulo se mencionan brevemente los subpaquetes **java.util**.

StringTokenizer

El procesamiento de un texto a menudo comienza con el *análisis sintáctico* de una cadena formateada. Este análisis implica la división del texto en un conjunto de partes, llamadas *tokens*, que en cierto orden tienen un significado semántico. La clase **StringTokenizer** facilita el primer paso en el proceso de análisis, a veces llamado *análisis léxico*. La clase **StringTokenizer** implementa la interfaz **Enumeration**. Por tanto, dada una cadena de entrada, se puede enumerar los tokens individuales contenidos en ella utilizando **StringTokenizer**.

Para utilizar **StringTokenizer** hay que especificar una cadena de entrada y una cadena que contenga delimitadores. Los *delimitadores* son caracteres que separan tokens. Cada carácter en la cadena de delimitadores se considera un delimitador válido. Por ejemplo, **",;: "** establece que los delimitadores son la coma, el punto y coma y los dos puntos. El conjunto por omisión de los delimitadores consiste en los caracteres de espacio en blanco: espacio, tabulación, nueva línea y retorno de carro.

Los constructores de **StringTokenizer** son los siguientes:

StringTokenizer(String *str*)
StringTokenizer(String *str*, String *delimitadores*)
StringTokenizer(String *str*, String *delimitadores*, boolean *delimToken*)

En las tres versiones, *str* es la cadena a tokenizar. En la primera, se utilizan los delimitadores por omisión. En la segunda y tercera, delimitadores es una cadena que indica los *delimitadores*. En la tercera versión, si *delimToken* es **true**, entonces los delimitadores también se devuelven como tokens al analizar la cadena. De lo contrario, los delimitadores no se devuelven. Las dos primeras formas no devuelven los delimitadores como tokens.

Una vez creado un objeto **StringTokenizer**, se utiliza el método **nextToken( )** para extraer tokens consecutivos. El método **hasMoreTokens( )** devuelve **true** mientras sigan existiendo más tokens a extraer. Dado que **StringTokenizer** implementa **Enumeration**, los métodos **hasMoreElements( )** y **nextElement( )** también se implementan, y pueden funcionar igual que **hasMoreTokens( )** y **nextToken( )** respectivamente. Los métodos de **StringTokenizer** se muestran en la Tabla 18-1.

Aquí hay un ejemplo que crea un **StringTokenizer** para analizar pares "clave=valor". Los pares "clave=valor" consecutivos están separados por punto y coma.

```
// Ejemplo de StringTokenizer.
import java.util.StringTokenizer;

class STDemo {
  static String in = "título=Java: Manual de Referencia;" +
    "autor=Schildt;" +
    "editorial=McGraw-Hill;" +
    "copyright=2009";

  public static void main(String args[]) {
    StringTokenizer st = new StringTokenizer(in, "=;");

    while(st.hasMoreTokens()) (
      String clave = st.nextToken();
      String val = st.nextToken();
      System.out.println(clave + "\t" + val);
    }
  }
}
```

La salida de este programa es ésta:

```
título Java: Manual de Referencia
autor Schildt
editorial McGraw-Hill
copyright 2009
```

Método	Descripción
int countTokens( )	Utilizando el conjunto actual de delimitadores, el método determina el número de tokens que quedan por analizar y devuelve el resultado.
boolean hasMoreElements( )	Devuelve **true** si quedan en la cadena uno o más tokens, y devuelve **false** si no hay ninguno.
boolean hasMoreTokens( )	Devuelve **true** si quedan en la cadena uno o más tokens, y devuelve **false** si no hay ninguno.
Object nextElement( )	Devuelve el siguiente token como un **Object**.
String nextToken( )	Devuelve el siguiente token como una **cadena**.
String nextToken(String *delim*)	Devuelve el siguiente token como una cadena y establece a *delim* como la cadena de *delimitadores*.

TABLA 18-1 Los métodos definidos por la clase **StringTokenizer**

BitSet

La clase **BitSet** crea un tipo especial de arreglo que contiene valores de bit. Este arreglo puede aumentar de tamaño según se necesite. Esto lo hace similar a un vector de bits. Los constructores de la clase BitSet son los siguientes:

BitSet( )
BitSet(int tamaño)

La primera versión crea un objeto por omisión. La segunda versión permite especificar su tamaño inicial (esto es, el número de bits que puede contener). Todos los bits se inicializan a cero.

La clase **BitSet** define los métodos listados en la Tabla 18-2.

Método	Descripción
void and(BitSet *conjuntoBits*)	Realiza una operación AND entre los contenidos del objeto **BitSet** que invoca y los especificados por el objeto *conjuntoBits*. El resultado se coloca en el objeto que invoca.
void andNot(BitSet *conjuntoBits*)	Para cada bit con valor 1 en el parámetro *conjuntoBits*, se limpia el correspondiente bit en el objeto **BitSet** que invoca.
void cardinality( )	Devuelve el número de bits en el conjunto que invoca.
void clear( )	Pone en cero todos los bits del objeto que invoca.
void clear(int *i*)	Pone en cero el bit especificado por la posición *i*.
void clear(int *inicio*, int *fin*)	Pone en cero los bits entre las posiciones *inicio* y *fin* –1.
Object clone( )	Duplica el objeto **BitSet** que invoca.
boolean equals(Object *conjuntoBits*)	Devuelve **true** si el conjunto de bits que invoca es equivalente al pasado en *conjuntoBits*. De lo contrario, el método devuelve **false**.
void flip(int *i*)	Invierte el bit especificado por *i*.
void flip(int *inicio*, int *fin*)	Invierte cada uno de los bits entre las posiciones *inicio* y *fin* –1.
boolean get(int *i*)	Devuelve el estado actual del bit en el índice especificado por *i*.
BitSet get(int *inicio*, int *fin*)	Devuelve un **BitSet** que contiene los bits desde la posición *inicio* hasta la posición *fin* –1 del objeto que invoca. El objeto que invoca no se modifica.
int hashCode( )	Devuelve el código de dispersión del objeto que invoca.
boolean intersects(BitSet *b*)	Devuelve **true** si al menos un par de bit correspondientes en el objeto que invoca y en el objeto *b* son 1.
boolean isEmpty( )	Devuelve **true** si todos los bits en el objeto que invoca son cero.
int length( )	Devuelve el número de bits requeridos para almacenar a los bits contenidos en el objeto **BitSet** que invoca. Este valor es determinado por la posición del último bit en 1.
int nextClearBit (int *i*)	Devuelve el índice del siguiente bit limpiado (esto es, el siguiente bit en cero), comenzando desde el índice especificado por *i*.

TABLA 18-2 Los métodos definidos por **BitSet**

Método	Descripción
int nextSetBit(int *inicio*)	Devuelve el índice del siguiente bit del conjunto (esto es, el siguiente bit en 1), comenzando con el índice especificado por el parámetro *inicio*. Si ningún bit está en 1, se devuelve –1.
void or(BitSet *conjuntoBits*)	Hace una operación OR de los contenidos del objeto **BitSet** que invoca con los del objeto especificado por *conjuntoBits*. El resultado se coloca en el objeto que invoca.
void set(int *i*)	Pone en 1 el bit especificado por el parámetro *i*.
void set(int *i*, boolean *v*)	Coloca el valor especificado por *v* en el bit especificado por *i*. El valor **true** pone un 1 en el bit.
void set(int *inicio*, int *fin*)	Pone en 1 los bits localizados desde *inicio* hasta *fin* –1.
void set(int *inicio*, int *fin*, boolean *v*)	Pone el valor especificado por *v* en los bits localizados desde *inicio* hasta *fin* –1. El valor **true** pone un 1 en el bit.
int size( )	Devuelve el número de bits en el objeto **BitSet** que invoca.
String toString( )	Devuelve la cadena equivalente al objeto **BitSet** que invoca.
void xor(BitSet *conjuntoBits*)	Hace una operación XOR de los contenidos del objeto **BitSet** que invoca con el especificado por *conjuntoBits*. El resultado se coloca en el objeto que invoca.

TABLA 18-2 Los métodos definidos por **BitSet** (*continuación*)

El siguiente ejemplo ilustra **BitSet**:

```
// Ejemplo con BitSet.
import java.util.BitSet;

class BitSetDemo {
  public static void main(String args[]) {
    BitSet bitsl = new BitSet(l6);
    BitSet bits2 = new BitSet(16);

    // dar valor a algunos bits
    for(int i=0; i<l6; i++) (
      if((i%2) == 0) bitsl.set(i);
      if((i%5) != 0) bits2.set(i);
    }

    System.out.println("Patrón inicial en bitsl: ");
    System.out.println(bitsl);
    System.out.println("\nPatrón inicial en bits2: ");
    System.out.println(bits2);

    // operación AND entre bits1 y bits2
    bits2.and(bitsl);
    System.out.println("\nbits2 AND bitsl: ");
    System.out.println(bits2);

    // operación OR entre bits1 y bits2
    bits2.or (bitsl);
    System.out.println("\nbits2 OR bitsl: ");
    System.out.println(bits2);
```

```
    // operación XOR entre bits1 y bits2
    bits2.xor(bits1);
    System.out.println("\nbits2 XOR bits1: ");
    System.out.println(bits2);
  }
}
```

La salida de este programa se muestra a continuación. Cuando **toString( )** convierte un objeto **BitSet** en su cadena equivalente, cada bit del conjunto se representa por su posición de bit. Los bits limpios (en cero) no se muestran.

```
Patrón inicial en bits1:
{0, 2, 4, 6, 8, 10, 12, 14}

Patrón inicial en bits2:
{1, 2, 3, 4, 6, 7, 8, 9, 11, 12, 13, 14}

bits2 AND bits1:
{2, 4, 6, 8, 12, 14}

bits2 OR bits1:
{0, 2, 4, 6, 8, 10, 12, 14}

bits2 XOR bits1:
{}
```


Date

La clase **Date** encapsula la fecha y hora actuales. Antes de empezar a examinar la clase **Date**, es importante indicar que ha cambiado sustancialmente desde su versión original definida por Java 1.0. Cuando salió Java 1.1, muchas de las funciones contenidas en la clase **Date** original se trasladaron a las clases **Calendar** y **DateFormat**, como resultado, muchos de los métodos de la clase original Date 1.0 fueron desechados. Dado que los métodos desechados no se deben utilizar en códigos nuevos, no los describiremos aquí.

La clase **Date** soporta los siguientes constructores:

Date( )
Date(long *miliseg*)

El primero inicializa el objeto con la fecha y hora actuales. El segundo constructor acepta un argumento que es el número de milisegundos transcurridos desde la medianoche del 1 de enero de 1970. Los métodos no descartados definidos por **Date** se muestran en la Tabla 18-3. La clase **Date** también implementa la interfaz **Comparable**.

Método	Descripción
boolean after(Date *fecha*)	Devuelve **true** si el objeto **Date** que invoca contiene una fecha que es posterior a la especificada por *fecha*. De lo contrario, devuelve **false**.
boolean before(Date *fecha*)	Devuelve **true** si el objeto **Date** que invoca contiene una fecha que es anterior a la especificada por *fecha*. De lo contrario, devuelve **false**.

TABLA 18-3 Los métodos en uso definidos por la clase **Date**

Método	Descripción
Object clone( )	Duplica el objeto **Date** que invoca.
int compareTo(Date *fecha*)	Compara el valor del objeto que invoca con el de *fecha*. Devuelve 0 si los valores son iguales. Devuelve un valor negativo si el objeto que invoca es anterior a *fecha*. Devuelve un valor positivo si el objeto que invoca es posterior a fecha.
boolean equals(Object *fecha*)	Devuelve **true** si el objeto **Date** que invoca contiene la misma fecha y hora que el especificado por *fecha*. De lo contrario, devuelve **false**.
long getTime( )	Devuelve el número de milisegundos transcurridos desde el 1 de enero de 1970.
int hashCode( )	Devuelve un código de dispersión para el objeto que invoca.
void. setTime(long *hora*)	Establece la fecha y hora según especifica *hora*, que representa el tiempo transcurrido en milisegundos desde la medianoche del 1 de enero de 1970.
String toString( )	Convierte el objeto Date que invoca en una cadena y devuelve el resultado.

TABLA 18-3 Los métodos en uso definidos por la clase **Date** (*continuación*)

Como se puede ver examinando la Tabla 18-3, las características de Date no permiten obtener los componentes individuales de la fecha o la hora. Como muestra el siguiente programa, sólo se puede obtener la fecha y la hora en forma de milisegundos o en su representación por omisión como cadena tal como la devuelve el método **toString( )**. Para obtener información más detallada sobre la fecha y la hora se utiliza la clase **Calendar**.

```
// Mostrar fecha y hora utilizando sólo métodos de la clase Date.
import java.util.Date;

class DateDemo {
  public static void main(String args[]) {
    // Instanciar un objeto Date
    Date date = new Date();

    // mostrar fecha y hora usando toString()
    System.out.println(date);

    // Mostrar el número de milisegundos desde medianoche del 01/01/1970 GMT
    long msec = date.getTime();
    System.out.println("Milisegundos desde 01/01/1970 GMT = " + msec);
  }
}
```

Una posible salida es:

```
Thu Jan 25 15:06:40 CEST 2001
Milisegundos desde 01/01/1970 GMT = 980456763420
```

Calendar

La clase abstracta **Calendar** proporciona un conjunto de métodos que permiten convertir un tiempo expresado en milisegundos en varios componentes útiles. Algunos ejemplos del tipo de información que se puede obtener son: año, mes, día, hora, minuto y segundo. Se pretende que las subclases de **Calendar** proporcionen una funcionalidad específica para interpretar la información del tiempo de acuerdo con sus propias reglas. Éste es un aspecto de la biblioteca de clases de Java que permite escribir programas que pueden operar en distintos entornos internacionales. Un ejemplo de esas subclases es **GregorianCalendar**.

Calendar no incluye constructores públicos.

Calendar define varias variables de instancia protegidas. **areFieldsSet** es una variable de tipo **boolean** que indica si se ha dado valor a las componentes del tiempo. **fields** es un arreglo de elementos tipo **int** que contiene los componentes de la hora. **isSet** es un arreglo de tipo **boolean** que indica si se ha dado valor a un componente específico de la hora. **time** es de tipo long, y contiene la hora actual para este objeto. **isTimeSet** es una variable de tipo **boolean** e indica si se ha establecido la hora actual.

Algunos métodos comúnmente utilizados definidos por **Calendar** se muestran en la Tabla 18-4.

Método	Descripción
abstract void add(int *f*, int *val*)	Añade *val* al componente de fecha y hora especificada por *f*. Para restar, se añade un valor negativo. El argumento *f* debe ser uno de los campos definidos por **Calendar**, por ejemplo **Calendar.HOUR.**
boolean after(Object *objCalendar*)	Devuelve **true** si el objeto **Calendar** que invoca contiene una fecha que es posterior a la especificada por *objCalendar*. De lo contrario, devuelve **false**.
boolean before(Object *objCalendar*)	Devuelve **true** si el objeto **Calendar** que invoca contiene una fecha que es anterior a la especificada por *objCalendar*. De lo contrario, devuelve **false**.
final void clear( )	Pone en cero todos los componentes de tiempo del objeto que invoca.
final void clear(int *c*)	Pone en cero la componente de tiempo especificada por *c* en el objeto que invoca.
Object clone( )	Devuelve un duplicado del objeto que invoca.
boolean equals(Object *objCalendar*)	Devuelve **true** si el objeto **Calendar** que invoca contiene una fecha que es igual a la especificada por *objCalendar*. De lo contrario, devuelve **false**.
int get(int *campoCalendar*)	Devuelve el valor de una componente del objeto que invoca. La componente se indica por *campoCalendar*. Ejemplos de las componentes que se pueden pedir son **Calendar.YEAR**, **Calendar.MONTH**, **Calendar.MINUTE**, etc.

TABLA 18-4 Métodos comúnmente utilizados definidos por la clase **Calendar**

Método	Descripción
static Locale[ ] getAvailableLocales( )	Devuelve una matriz de objetos **Locale** que contiene los ámbitos locales para los que hay calendarios disponibles.
static Calendar getInstance( )	Devuelve un objeto **Calendar** para la localidad y zona horaria por omisión.
static Calendar getInstance (TimeZone *zh*)	Devuelve un objeto **Calendar** para la zona horaria especificada por *zh*. Se utiliza la localidad por omisión.
static Calendar getInstance(Locale *local*)	Devuelve un objeto **Calendar** para la localidad especificada por *local*. Se utiliza la zona horaria por omisión.
static Calendar getInstance (TimeZone zh, Locale *local*)	Devuelve un objeto **Calendar** para la zona horaria especificada por *zh* y la localización especificada por *local*.
final Date getTime( )	Devuelve un objeto **Date** equivalente a la hora del objeto que invoca.
TimeZone getTimeZone( )	Devuelve la zona horaria para el objeto que invoca.
final boolean isSet(int *c*)	Devuelve **true** si el componente especificado tiene un valor. De lo contrario, devuelve **false**.
void set(int *c*, int *val*)	Establece la componente de fecha u hora especificada por *c* al valor especificado por *val* en el objeto que invoca. El argumento *c* debe ser uno de los campos definidos por **Calendar**, como **Calendar.HOUR**.
final void set(int *año*, int *mes*, int *diaDelMes*)	Establece varias componentes de fecha y hora en el objeto que invoca.
final void set(int *año*, int *mes*, int *diaDelMes*, int *horas*, int *minutos*)	Establece varias componentes de fecha y hora en el objeto que invoca.
final void set(int *año*, int *mes*, int *diaDelMes*, int *horas*, int *minutos*, int *segundos*)	Establece varias componentes de fecha y hora en el objeto que invoca.
final void setTime(Date *d*)	Establece varias componentes de fecha y hora en el objeto que invoca. Esta información se obtiene del objeto **Date** *d*.
void setTimeZone(TimeZone *zh*)	Establece la zona horaria para el objeto que invoca a la especificada por *zh*.

TABLA 18-4 Métodos comúnmente utilizados definidos por la clase **Calendar** (*continuación*)

Calendar define las siguientes constantes **enteras**, que se utilizan cuando se obtienen o establecen componentes del calendario:

ALL_STYLES	FRIDAY	PM
AM	HOUR	SATURDAY
AM_PM	HOUR_OF _DAY	SECOND
APRIL	JANUARY	SEPTEMBER
AUGUST	JULY	SHORT
DATE	JUNE	SUNDAY
DAY_OF_MONTH	LONG	THURSDAY
DAY_OF_WEEK	MARCH	TUESDAY
DAY_OF_WEEK_IN_MONTH	MAY	UNDECIMBER
DAY_OF_YEAR	MILLISECOND	WEDNESDAY
DECEMBER	MINUTE	WEEK_OF _MONTH
DST_OFFSET	MONDAY	WEEK_OF_YEAR
ERA	MONTH	YEAR
FEBRUARY	NOVEMBER	ZONE_OFFSET
FIELD_COUNT	OCTOBER	

El siguiente programa ilustra diferentes métodos de **Calendar**:

```
// Ejemplos de métodos de la clase Calendar
import java.util.Calendar;

class CalendarDemo {
  public static void main(String args[]) {
    String months [] = {
             "Ene", "Feb", "Mar", "Abr",
             "May", "Jun", "Jul", "Ago",
             "Sep", "Oct", "Nov", "Dic"};

    // Crear un calendario inicializado con la
    // fecha y la hora actuales en el ámbito de
    // localidad y zona horaria por omisión.
    Calendar calendar = Calendar.getlnstance();

    // Mostrar información de fecha y hora actuales.
    System.out.print("Fecha: ");
    System.out.print(months[calendar.get(Calendar.MONTH)]);
    System.out.print(" " + calendar.get(Calendar.DATE) + " ");
    System.out.println(calendar.get(Calendar.YEAR));

    System.out.print("Hora: ");
    System.out.print(calendar.get(Calendar.HOUR) + ":");
    System.out.print(calendar.get (Calendar.MINUTE) + ":");
    System.out.println(calendar.get(Calendar.SECOND));

    // Establecer la información de fecha y hora y mostrarla.
    calendar.set(Calendar.HOUR, 10);
    calendar.set(Calendar.MINUTE, 29);
    calendar.set(Calendar.SECOND, 22);
```

```
    System.out.print("Hora actualizada: ");
    System.out.print(calendar.get(Calendar.HOUR) + ":");
    System.out.print(calendar.get(Calendar.MINUTE) + ":");
    System.out.println(calendar.get(Calendar.SECOND));
  }
}
```

Un ejemplo de salida es:

```
Fecha: Jan 1 2007
Hora: 11:24:25
Hora actualizada: 10:29:22
```

GregorianCalendar

La clase **GregorianCalendar** es una implementación concreta de **Calendar** que implementa el calendario gregoriano común, con el que estamos familiarizados. El método **getInstance( )** de Calendar devuelve un **GregorianCalendar** inicializado con la fecha y la hora actuales en el ámbito de la localidad y zona horaria por omisión.

GregorianCalendar define dos campos: **AD** y **BC**. Estos campos representan las dos eras definidas por el calendario gregoriano.

También existen diversos constructores para objetos **GregorianCalendar**. El constructor por omisión, **GregorianCalendar( )**, inicializa el objeto con la fecha y hora actuales en la localidad y zona horaria por omisión. Otros tres constructores ofrecen mayores niveles de especificidad:

GregorianCalendar(int *año*, int *mes*, int *diaDelMes*)
GregorianCalendar(int *año*, int *mes*, int *diaDelMes*, int *horas*,
int *minutos*)
GregorianCalendar(int *año*, int *mes*, int *diaDelMes*, int *horas*,
int *minutos*, int *segundos*)

Las tres versiones establecen día, mes y año. Aquí, *año* indica el año. El mes se especifica en *mes*, con cero indicando enero. El día del mes se especifica en *diaDelMes*. La primera versión pone la hora en medianoche. La segunda también establece las horas y los minutos. La tercera versión añade segundos.

También se puede construir un objeto **GregorianCalendar** especificando la localidad y la zona horaria. Los siguientes constructores crean objetos inicializados con la fecha y hora actuales usando la zona horaria y localidad especificados:

GregorianCalendar(Locale *local*)
GregorianCalendar(TimeZone *zonaHoraria*)
GregorianCalendar(TimeZone *zonaHoraria*, Locale *local*)

GregorianCalendar proporciona una implementación de todos los métodos abstractos de **Calendar**. También proporciona algunos métodos adicionales. Quizá el más interesante sea **isLeapYear( )**, que comprueba si un año es bisiesto. Su forma es:

boolean isLeapYear(int *año*)

Este método devuelve **true** si *año* es bisiesto, y **false** si no.

El siguiente programa ejemplifica el uso de **GregorianCalendar**:

```
// Ejemplo de GregorianCalendar
import java.util.*;

class GregorianCalendarDemo {
  public static void main(String args[]) {
    String months[] = {
              "Ene", "Feb", "Mar", "Abr",
              "May", "Jun", "Jul", "Ago",
              "Sep", "Qct", "Nov", "Dic"};
    int año;

    // Crear un calendario gregoriano inicializado
    // con la fecha y hora actuales en la
    // localidad y zona horaria por omisión.
    GregorianCalendar gcalendar = new GregorianCalendar();

    // Mostrar fecha y hora actuales.
    System.out.print("Fecha: "),
    System.out.print(months[gcalendar.get(Calendar.MONTH)]);
    System.out.print(" " + gcalendar.get(Calendar.DATE) + " ");
    System.out.println(año = gcalendar.get(Calendar.YEAR));

    System.out.print("Hora: ");
    System.out.print(gcalendar.get(Calendar.HOUR) + ":");
    System.out.print (gcalendar.get (Calendar.MINUTE) + ":");
    System.out.println(gcalendar.get(Calendar.SECOND)); ,

    // Comprobar si el año actual es bisiesto
    if(gcalendar.isLeapYear(año)) (
      System.out.println("El año actual es bisiesto"),
    }
    else {
      System.out.println("El año actual no es bisiesto");
    }
  }
}
```

Una posible salida es ésta:

```
Fecha: Jan 1 2007
Hora: 11:25:27
El año actual no es bisiesto
```

TimeZone

Otra clase relacionada con el tiempo es **TimeZone**. La clase **TimeZone** permite trabajar con desplazamientos de zonas horarias respecto a la hora del meridiano de Greenwich (GMT por sus siglas en inglés), también llamado Tiempo Universal Coordinado (UTC por sus siglas en inglés). También calcula el cambio de horario de verano. **TimeZone** sólo proporciona el constructor por omisión.

Algunos métodos definidos por **TimeZone** se resumen en la Tabla 18-5.

Método	Descripción
Object clone( )	Devuelve una versión de **clone( )** específica para **TimeZone**.
static String[] getAvailableIDs( )	Devuelve un arreglo de objetos tipo **String** que representan los nombres de todas las zonas horarias.
static String[] getAvailableIDs(int *delta*)	Devuelve un arreglo de objetos tipo **String** que representan los nombres de todas las zonas horarias desplazadas en delta respecto a GMT.
static TimeZone getDefault( )	Devuelve un objeto **TimeZone** que representa la zona horaria por omisión utilizada en la computadora donde se ejecuta la instrucción.
String getID( )	Devuelve el nombre del objeto **TimeZone** que invoca.
abstract int getOffset(int *era*, int *año*, int *mes*, int *diaMes*, int *diaSemana*, int *milisegundos*)	Devuelve el desplazamiento que debería añadirse a GMT para calcular el tiempo local. Este valor se ajusta para el horario de verano. Los parámetros pasados al método representan componentes de fecha y hora.
abstract int getRawOffset( )	Devuelve el desplazamiento bruto que debería añadirse a GMT para calcular la hora local. Este valor no se ajusta para el horario de verano.
static TimeZone getTimeZone (String *zh*)	Devuelve el objeto **TimeZone** de la zona horaria llamada *zh*.
abstract boolean inDaylightTime(Date *d*)	Devuelve **true** si la fecha representada por *d* está en horario de verano en el objeto que invoca. De lo contrario, devuelve **false**.
static void setDefault(TimeZone *zh*)	Establece la zona horaria a utilizar en este nodo. *zh* es una referencia para el objeto **TimeZone** que ha de usarse.
void setID(String *nombreZh*)	Establece el nombre de la zona horaria (esto es, su ID) al especificado por *nombreZh*.
abstract void setRawOffset (int *milisegundos*)	Establece la diferencia respecto al tiempo GMT en *milisegundos*.
abstract boolean useDaylightTime( )	Devuelve **true** si el objeto que invoca utiliza horario de ahorro de luz diurna. De lo contrario, devuelve **false**.

TABLA 18-5 Algunos de los métodos definidos por **TimeZone**

SimpleTimeZone

La clase **SimpleTimeZone** es una subclase útil de **TimeZone**. Implementa los métodos abstractos de **TimeZone** y permite trabajar con zonas horarias y un calendario gregoriano. También calcula el cambio de horario para aprovechar la luz diurna.

SimpleTimeZone define cuatro constructores. Uno de éstos es:

SimpleTimeZone(int *delta,* String *nombreZh*)

El constructor crea un objeto **SimpleTimeZone**. El desplazamiento respecto al tiempo GMT se da en *delta*. El nombre de la zona horaria es *nombreZh*.

El segundo constructor de **SimpleTimeZone** es:

SimpleTimeZone(int *delta*, String *idZh*, int *halMes0*,
int *halDiaDelMes0*, int *halDia0*, int *hora0*,
int *halMesl*, int *halDiaDelMesl*, int *halDial*,
int *horal*)

Aquí, el desplazamiento relativo a la hora GMT se especifica en *delta*. El nombre de la zona horaria se pasa en *idZh*. El comienzo del horario de verano se indica en los parámetros *halMes0*, *halDiaDelMes0*, *halDia0* y *hora0*. El final del horario de verano se indica en los parámetros *halMes1*, *halDiaDelMes1*, *halDia1* y *hora1*.

El tercer constructor de **SimpleTimeZone** es:

SimpleTimeZone(int *delta*, String *idZh*, int *halMes0*,
int *halDiaDelMes0*, int *halDia0*, int *hora0*,
int *halMes1*, int *halDiaDelMes1*, int *halDia1*,
int *hora1*, int *halDelta*) .

Aquí, *halDelta* es el número de milisegundos ahorrados durante el horario de verano.
El cuarto constructor de **SimpleTimeZone** es:

SimpleTimeZone(int *delta*, String *idZh*, int *halMes0*,
int *halDiaDelMes0*, int *halDia0*, int *hora0*,
int *modo0*, int *halMes1*, int *halDiaDelMes1*, int *halDía1*,
int *hora1*, int *modo1*, int *halDelta*) .

Aquí, *modo0* especifica el modo del tiempo inicial, y *modo1* especifica el modo del tiempo final. Valores válidos para los modos incluyen:

STANDARD_TIME	WALL_TIME	UTC_TIME

El modo de tiempo indica como los valores de tiempo son interpretados. El modo por omisión es **WALL_TIME**.

Locale

La clase **Locale** se instancia para producir objetos que describen regiones geográficas y culturales. Es una de las varias clases que permiten escribir programas que se puedan ejecutar en ámbitos internacionales diferentes. Por ejemplo, los formatos utilizados para mostrar fechas, horas y números son diferentes en distintas regiones.

La internacionalización es un vasto campo que está fuera del alcance de este libro. Sin embargo, la mayoría de los programas sólo necesitan trabajar con los aspectos básicos, lo que incluye establecer un objeto **Locale**.

La clase **Locale** define las siguientes constantes, que permiten trabajar con las localidades más comunes:

CANADA	GERMAN	KOREA
CANADA_FRENCH	GERMANY	PRC

CHINA	ITALIAN	SIMPLIFIED _CHINESE
CHINESE	ITALY	TAIWAN
ENGLISH	JAPAN	TRADITIONAL_CHINESE
FRANCE	JAPANESE	UK
FRENCH	KOREAN	US

Por ejemplo, la expresión **Locale.CANADA** representa al objeto **Locale** para Canadá.

Los constructores de la clase **Locale** son:

Locale(String *lenguaje*)
Locale(String *lenguaje*, String *pais*)
Locale(String *lenguaje*, String *pais*, String *datos*)

Estos constructores crean un objeto **Locale** que representa un *lenguaje* específico y en el caso de los últimos dos un *país* específico. Estos valores deben contener códigos de lenguaje y país especificados con el estándar ISO. En *datos* se puede proporcionar información auxiliar sobre el navegador o información específica sobre el vendedor.

Locale define varios métodos. Uno de los más importantes es **setDefault( )** definido como:

static void setDefault(Locale *objLocale*)

Esto establece la localidad por omisión a la indicada en *objLocale*.

Algunos otros métodos interesantes de la clase son los siguientes:

final String getDisplayCountry( )
final String getDisplayLanguage( )
final String getDisplayName( )

Estos métodos devuelven cadenas legibles que se pueden utilizar para mostrar el nombre del país, el lenguaje, y la descripción completa de la localidad.

El objeto Locale por omisión se puede obtener usando el método **getDefault( )**:

static Locale getDefault( )

Calendar y **GregorianCalendar** son ejemplos de clases que operan en forma sensible a la localidad. **DateFormat** y **SimpleDateFormat** también dependen de la localidad.

Random

La clase **Random** es un generador de números *pseudoaleatorios*. Estos números se llaman así porque son simplemente sucesiones de números distribuidos uniformemente. **Random** define los siguientes constructores:

Random( )
Random(long *semilla*)

La primera versión crea un generador de números que usa la hora actual como valor de arranque, o *semilla*. La segunda forma permite especificar una semilla manualmente.

Método	Descripción
boolean nextBoolean( )	Devuelve el siguiente valor aleatorio de tipo **boolean**.
void nextBytes(byte *vals*[ ])	Llena *vals* con valores generados aleatoriamente.
double nextDouble( )	Devuelve el siguiente número aleatorio de tipo **double**.
float nextFloat( )	Devuelve el siguiente número aleatorio de tipo **float**.
double nextGaussian( )	Devuelve el siguiente número aleatorio **gaussiano**.
int nextInt( )	Devuelve el siguiente número aleatorio de tipo **int**.
int nextInt(int *n*)	Devuelve el siguiente número aleatorio de tipo **int** dentro del intervalo de cero a *n*.
long nextLong( )	Devuelve el siguiente número aleatorio **long**.
void setSeed(long *nuevaSemilla*)	Establece el valor de la semilla (esto eso, el punto de arranque para el generador de números aleatorios) al especificado por *nuevaSemilla*.

TABLA 18-6 Los métodos definidos por la clase **Random**

Si se inicializa un objeto **Random** con una semilla, se define el punto de arranque para la sucesión aleatoria. Si se usa la misma semilla para inicializar otro objeto **Random**, se conseguirá la misma sucesión aleatoria. Para generar sucesiones diferentes basta con especificar valores diferentes de la semilla. El modo más fácil de hacer esto es usar la hora actual como semilla de un objeto **Random**. Este modo de proceder reduce la posibilidad de obtener sucesiones repetidas.

Los métodos públicos definidos por **Random** se muestran en la Tabla 18-6.

Como se puede ver, hay siete tipos de números aleatorios que se pueden extraer de un objeto **Random**. Los valores **boleanos** aleatorios están disponibles mediante el método **nextBoolean( )**. Llamando a **nextBytes( )** se puede obtener bytes aleatorios. El método **nextInt( )** devuelve enteros. Con **nextLong( )** se obtienen enteros largos, uniformemente distribuidos sobre su rango. Los métodos **nextFloat( )** y **nextDouble( )** devuelven números **float** y **double**, respectivamente, uniformemente distribuidos entre 0.0 y 1.0. Finalmente, **nextGaussian( )** devuelve un valor **double** centrado en 0.0 con una desviación estándar de 1.0. Esto es, la curva conocida como *campana de Gauss.*

El ejemplo siguiente ilustra la sucesión producida por **nextGaussian( )**. Obtiene 100 números aleatorios gaussianos y calcula su media. El programa también cuenta el número de valores comprendidos entre dos desviaciones estándar, utilizando incrementos de 0.5 para cada categoría. El resultado se muestra gráficamente en la pantalla en forma de barras horizontales.

```
// Ejemplo con valores aleatorios gaussianos.
import java.util.Random;
class RandDemo {
  public static void main(String args[]) {
    Random r = new Random ();
    double val;
    double sum = 0;
    int campana[] = new int[10];

    for(int i=0; i<100; i++) {
      val = r.nextGaussian();
      sum += val;
      double t = -2;
```

```
      for(int x=0; x<10; x++, t += 0.5)
        if(va1 < t) {
        campana [x] ++;
        break;
      }
    }
    System.out.println("Media de los valores: " +
                       (sum/100));

    // mostrar la campana en forma de barras horizontales
    for(int i=0; i<10; i++) {
      for(int x = campana[i]; x>0; x--)
        System.out.print("*");
      System.out.println();
    }
  }
}
```

Aquí está la salida de una ejecución del programa. Como se ve, se obtiene una distribución de números con forma de campana.

```
Media de los valores: 0.0702235271133344
**
*******
******
***************
******************
*****************
*************
**********
********
***
```

Observable

La clase **Observable** se utiliza para crear subclases que otras partes del programa pueden observar. Cuando un objeto de esta subclase sufre un cambio, se notifica a las clases observadoras. Las clases observadoras deben implementar la interfaz **Observer**, que define al método **update( )**. El método **update( )** es llamado cuando a un observador se le notifica un cambio en un objeto observado.

Observable define los métodos de la Tabla 18-7. Un objeto que está siendo observado debe seguir dos sencillas reglas. Primero, si ha cambiado, debe llamar a **setChanged( )**. Segundo, cuando esté listo para notificar a los observadores de este cambio, debe llamar a **notifyObservers( )**. Esto causa que se llame a su vez al método **update( )** de los objetos observadores. Pero cuidado: si el objeto llama a **notifyObservers( )** sin haber llamado previamente a **setChanged( )**, no pasará nada. El objeto observado debe llamar tanto a **setChanged( )** como a **notifyObservers( )** antes de que se llame a **update( )**.

Nótese que **notifyObservers( )** tiene dos formas: una que toma un argumento y otra que no lo toma. Si se llama a **notifyObservers( )** con un argumento, el objeto se pasa al método **update( )** del observador como segundo parámetro. De lo contrario, se pasa **null** a **update( )**. Se puede utilizar el segundo parámetro para pasar cualquier tipo de objeto que sea apropiado para la aplicación que se realiza.

Método	Descripción
void addObserver(Observer *obj*)	Añadir *obj* a la lista de objetos que observan al objeto que invoca.
protected void clearChanged( )	Una llamada a este método devuelve el estado del objeto que invoca a "no cambiado".
int countObservers( )	Devuelve el número de objetos que observan al objeto que invoca.
void deleteObserver(Observer *obj*)	Quita *obj* de la lista de objetos que observan al objeto que invoca.
void deleteObservers( )	Quita todos los observadores del objeto que invoca.
boolean hasChanged( )	Devuelve **true** si el objeto que invoca ha sido modificado, y **false**, si no.
void notifyObservers( )	Notifica a todos los observadores del objeto que invoca que ha cambiado, llamando a **update( )**. Se pasa un **null** como segundo argumento a **update( )**.
void notifyObservers(Object *obj*)	Notifica a todos los observadores del objeto que invoca que ha cambiado, llamando a **update( )**. *obj* es pasado como argumento a **update( ).**
protected void setChanged( )	Es llamado cuando el objeto que invoca ha cambiado.

TABLA 18-7 Los métodos definidos por la clase **Observable**

La interfaz Observer

Para observar un objeto observable se debe implementar la interfaz **Observer**. Esta interfaz define únicamente el método mostrado a continuación:

void update(Observable *objObservado*, Object *arg)*

Aquí, *objObservado* es el objeto observado, y *arg* es el valor pasado por **notifyObservers( )**. El método **update( )** es llamado cuando tiene lugar un cambio en el objeto observado.

Un ejemplo con la interfaz Observer

Presentamos a continuación un ejemplo que ilustra el uso de un objeto observable. Este ejemplo crea una clase observadora, llamada **Mirador**, que implementa la interfaz **Observer**. La clase que es monitoreada se llama **Sujeto**, y extiende a **Observable**. Dentro de **Sujeto** está el método **contador( )**, que simplemente hace una cuenta regresiva desde un valor especificado. La clase utiliza **sleep( )** para esperar una décima de segundo en cada conteo. Cada vez que la cuenta cambia, se llama a **notifyObservers( )** pasando el estado actual de la cuenta como argumento. Esto hace llamar al método **update( )** dentro de **Mirador**, que muestra el estado actual de la cuenta. Dentro de **main( )** se crea un objeto de clase **Mirador** y un objeto de clase **Sujeto**, llamados respectivamente **observando** y **observado**. Entonces, **observando** se añade a la lista de observadores de **observado**. Esto significa que cada vez que **contador( )** llame a **notifyObservers( )**, se llamará a **observando.update( )**.

```
/*  Ejemplo de la interfaz Observer y
    una clase observada.
*/
```

```
import java.util.*;

// Esta es la clase observadora.
class Mirador implements Observer {
  public void update(Observable obj, Object arg) {
    System.out.println("update( ) llamado; la cuenta está en " +
                       ((Integer)arg).intValue());
  }
}

// Esta es la clase observada.
class Sujeto extends Observable {
  void contador(int period) {
    for( ; period >=0; period--) {
      setChanged( );
      notifyObservers(new Integer(period));
      try {
        Thread.sleep(100);
      } catch(InterruptedException e) {
        System.out.println("despertó el hilo");
      }
    }
  }
}

class ObserverDemo{
  public static void main(String args[]) {
    Sujeto observado = new Sujeto() ;
    Mirador observando = new Mirador();

    /*  Añadir observando a la lista de observadores
        para el objeto observado. */
    observado.addObserver(observando);

    observado. contador (10) ;
  }
}
```

La salida de este programa sería la siguiente:

```
update ( ) llamado; la cuenta está en 10
update ( ) llamado; la cuenta está en 9
update ( ) llamado; la cuenta está en 8
update ( ) llamado; la cuenta está en 7
update ( ) llamado; la cuenta está en 6
update ( ) llamado; la cuenta está en 5
update ( ) llamado; la cuenta está en 4
update ( ) llamado; la cuenta está en 3
update ( ) llamado; la cuenta está en 2
update ( ) llamado; la cuenta está en 1
update ( ) llamado; la cuenta está en 0
```

Más de un objeto puede ser un observador. Por ejemplo, el siguiente programa implementa dos clases observadoras y añade un objeto de cada clase a la lista de observadores de **Sujeto**. El segundo observador espera hasta que la cuenta llegue a cero y entonces hace sonar una campana.

```
/*  Un objeto puede ser observado por dos o más
    observadores.
*/

import java.util.*;

// Ésta es la primera clase observadora.
class Miradorl implements Observer {
  public void update(Observable obj, Object arg) {
    System.out.println("update() llamado; la cuenta está en " +
                     ((Integer)arg) .intValue());
  }
}

// Esta es la segunda clase observadora.
class Mirador2 implements Observer {
  public void update(Observable obj, Object arg) {
    // Hacer sonar la campana al terminar
    if(((Integer)arg).intValue() == 0)
      System.out.println("Hecho" + '\7');
  }
}

// Esta es la clase observada.
class Sujeto extends Observable {
  void contador(int period) {
    for( ; period >=0; period--) {
      setChanged () ;
      notifyObservers (new Integer (period) );
      try{
        Thread.sleep(l00);
      } catch(InterruptedException e) {
        System.out.println("despertó el hilo");
      }
    }
  }
}

class DosObservadores {
  public static void main(String args[]) {
    Sujeto observado = new Sujeto( );
    Miradorl observandol = new Miradorl ( );
    Mirador2 observando2 = new Mirador2( );

    // añadir ambos observadores
    observado.addObserver(observandol);
    observado. addObserver (observando2);

    observado.contador(l0);
  }
}
```

La clase **Observable** y la interfaz **Observer** permiten implementar sofisticadas arquitecturas de programación basadas en la metodología documento/vista. También son útiles en aplicaciones multihilo.

Timer y TimerTask

Una característica interesante y útil que ofrece el paquete **java.util** es la posibilidad de programar una tarea para su ejecución en cierto momento futuro. Las clases que soportan esto son **Timer** y **TimerTask**. Utilizando estas clases se puede crear un hilo que corra en segundo plano y que espere a determinado momento. Cuando el momento llega, la tarea vinculada a ese hilo se ejecuta. Hay varias opciones que permiten programar una tarea para su ejecución reiterada o en determinado momento. Aunque siempre es posible crear manualmente una tarea que se ejecute en un momento determinado utilizando la clase **Thread**, las clases **Timer** y **TimerTask** simplifican mucho este proceso.

Timer y **TimerTask** trabajan juntas. **Timer** es la clase que se utiliza para programar una tarea para su ejecución. La tarea programada debe ser una instancia de **TimerTask**. Así, para programar una tarea, primero se creará un objeto **TimerTask** y luego se programa su ejecución utilizando una instancia de **Timer**.

TimerTask implementa la interfaz **Runnable**; así puede utilizarse para crear un hilo de ejecución. Su constructor se muestra aquí:

TimerTask( )

TimerTask define los métodos de la Tabla 18-8. Nótese que **run( )** es abstracto, lo que significa que debe ser sobrescrito. El método **run( )**, definido por la interfaz **Runnable**, contiene el código que se ejecutará. Así, el modo más fácil de crear una tarea programada es extender **TimerTask** y sobrescribir **run( )**.

Una vez creada una tarea, se programa su ejecución por un objeto del tipo **Timer**. Los constructores de **Timer** son los siguientes:

Timer( )
Timer(boolean *demonio*)
Timer (String *nombre*)
Timer (String *nombre*, boolean *demonio*)

La primera versión crea un objeto **Timer** que corre como un hilo normal. La segunda utiliza un hilo de tipo *demonio* si el argumento demonio es **true**. Un hilo de tipo demonio se ejecutará sólo en tanto el resto del programa sigue ejecutándose. El tercer y cuarto constructor permiten especificar un nombre para el objeto **Timer**. Los métodos definidos por **Timer** se muestran en la Tabla 18-9.

Método	Descripción
boolean cancel( )	Termina la tarea. Devuelve **true** si la ejecución de la tarea ha sido evitada. De lo contrario se devuelve **false**.
abstract void run( )	Contiene el código de la tarea a realizar.
long scheduledExecutionTime( )	Devuelve la hora a la que estaba programado que tuviera lugar la última ejecución de la tarea.

TABLA 18-8 Los métodos definidos por **TimerTask**

Método	Descripción
void cancel( )	Cancela al hilo del temporizador.
int purge( )	Borra las tareas canceladas de la fila de tareas del temporizador.
void schedule(TimerTask *tarea*, long *espera*)	*tarea* está programada para su ejecución después de transcurrido el periodo *espera*. El parámetro *espera* se especifica en milisegundos.
void schedule(TimerTask *tarea*, long *espera*, long *repite*)	*tarea* está programada para su ejecución después de transcurrido el periodo *espera*. La *tarea* se ejecuta entonces repetidamente con el periodo especificado por *repite*. Tanto *espera* como *repite* se especifican en milisegundos.
void schedule(TimerTask *tarea*, Date *hora*)	*tarea* está programada para su ejecución en el momento especificado por *hora*.
void schedule(TimerTask *tarea*, Date *hora*, long *repite*)	*tarea* está programada para su ejecución en el momento especificado por *hora*. La *tarea* se ejecuta entonces repetidamente con el periodo pasado en *repite*. El parámetro *repite* se especifica en milisegundos.
void scheduleAtFixedRate (TimerTask *tarea*, long *espera*, long *repite*)	*tarea* está programada para su ejecución después de transcurrido el periodo de *espera*. La tarea se ejecuta entonces repetidamente con el periodo especificado por *repite*. Tanto *espera* como *repite* se especifican en milisegundos. El momento de cada repetición es relativo a la primera ejecución, no a la ejecución precedente. Por tanto, la frecuencia global de ejecución es fija.
void scheduleAtFixedRate (TimerTask *tarea*, Date *hora*, long *repite*)	*tarea* está programada para su ejecución a la hora especificada. La tarea se ejecuta entonces repetidamente con el periodo especificado por *repite*. El parámetro *repite* se especifica en milisegundos. El momento de cada repetición es relativo a la primera ejecución, no la ejecución precedente. Por tanto, la frecuencia global de ejecución es fija.

TABLA 18-9 Los métodos definidos por **Timer**

Una vez creado un **Timer**, se programa una tarea llamando a **schedule( )** sobre el **Timer** creado. Como muestra la Tabla 18-9, hay varias formas de **schedule( )** que permiten programar una tarea de diferentes maneras.

Si se crea una tarea que no es de tipo demonio, entonces se llamará a **cancel( )** para terminar la tarea cuando el programa termine. Si esto no se hace, el programa puede quedarse "colgado" por un periodo de tiempo.

El siguiente programa ejemplifica **Timer** y **TimerTask**. Define una tarea programada cuyo método **run( )** saca el mensaje: "Tarea programada ejecutada". Esta tarea se programa para correr una vez cada medio segundo después de una espera inicial de un segundo.

```
// Ejemplo con Timer y TimerTask.

import java.util.*;

class MiTimerTask extends TimerTask {

  public void run( ) {
    System.out.println("Tarea programada ejecutada.");
  }
}

class TTest {
  public static void main(String args[]) {
    MiTimerTask miTask = new MiTimerTask();
    Timer miTimer = new Timer();

    /*  Establecer una espera inicial de 1 segundo,
        y luego repetir cada medio segundo.
    */
    miTimer.schedule(miTask, 1000, 500);

    try {
      Thread.sleep(5000);
    } catch (InterruptedException exc) {}

    miTimer.cancel( );
  }
}
```

Currency

La clase **Currency** encapsula información acerca de una moneda o divisa. No define constructores. Los métodos definidos por la clase **Currency** se muestran en la Tabla 18-10. El siguiente programa es un ejemplo de **Currency**:

```
// Ejemplo de Currency
import java.util.*;

class CurDemo{
  public static void main(String args[]) {
    Currency c;

    c = Currency.getInstance(Locale.US)

    System.out.println("Símbolo:" + c.getSymbol());
    System.out.println("Dígitos fraccionarios por omisión: " +
                       c.getDefaultFractionDigits());
  }
}
```

La salida del programa se muestra a continuación:

```
Símbolo : $
Dígitos fraccionarios por omisión: 2
```

Método	Descripción
String getCurrencyCode( )	Devuelve el código (definido por el estándar ISO 4217) que describe a la divisa que invoca.
int getDefaultFractionDigits( )	Devuelve el número de dígitos después del punto decimal que se emplean normalmente con la divisa que invoca. Por ejemplo, con el dólar normalmente se emplean dos dígitos decimales.
static Currency getInstance(Locale *obj*)	Devuelve un objeto **Currency** para la localidad especificada en *obj*.
static Currency getInstance(String *c*)	Devuelve al objeto **Currency** asociado con el código de divisa dado en el argumento *c*.
String getSymbol( )	Devuelve el símbolo de la divisa (por ejemplo $) para el objeto que invoca.
String getSymbol(Locale *obj*)	Devuelve el símbolo de la divisa (por ejemplo $) para la localidad especificada por el objeto *obj*.
String toString( )	Devuelve el código de la divisa que invoca.

TABLA 18-10 Los métodos definidos por la clase **Currency**

Formatter

Con la liberación del JDK 5, Java añadió una característica largamente esperada por los programadores: la habilidad de crear fácilmente salidas con formato. Desde sus inicios, Java ha ofrecido una rica y variada API, sin embargo no siempre ha ofrecido una forma simple de crear una salida de texto con formato, especialmente para valores numéricos. Clases como **NumberFormat**, **DateFormat** y **MessageFormat** disponibles en las versiones previas de Java contaban con características de formato muy útiles, pero no eran del todo cómodo su uso. Además, a diferencia de C y C++, Java no ofrecía una familia de funciones equivalente a la ampliamente conocida y utilizada **printf( )**; la cual ofrece una forma simple de dar formato a la salida. Una razón por la cual Java no ofrecía esta funcionalidad es el hecho de que los métodos estilo **printf** requieren el uso de argumentos de longitud variable (varargs), los cuales fueron soportados por Java hasta la aparición del JDK 5. Una vez que los argumentos de longitud variable estuvieron disponibles fue simple añadir formateadotes de propósito general.

En el núcleo del soporte que Java provee para formatear salida, se encuentra la clase **Formatter**. Esta clase proporciona las *conversiones de formato* que permiten desplegar números, texto, hora y fecha en prácticamente cualquier formato que se desee. Esta funcionalidad trabaja en Java de forma similar a como lo hace la función **printf( )** de C y C++, lo cual significa que si el programador está familiarizado con C / C++, entonces aprender a utilizar la clase **Formatter** es realmente fácil. Si no se está familiarizado con C / C++ de igual forma es muy sencillo darle formato a los datos.

NOTA *Aunque la clase **Formatter** de Java funciona de forma muy similar al método **printf( )** de C y C++, existen algunas diferencias y algunas características nuevas. Por tanto, si el programador tiene un conocimiento previo de C / C++ se recomienda una lectura cuidadosa.*

Constructores de la clase Formatter

Antes de utilizar la clase **Formatter** para darle formato a un texto debemos crear un objeto de tipo **Formatter**. En general, el trabajo realizado por **Formatter** consiste en convertir la forma binaria de los datos empleados por un programa en un texto con formato. Almacenando el texto formateado en un búfer, el contenido del cual puede ser obtenido por el programa siempre que sea necesario. Es posible que **Formatter** provea este búfer automáticamente, o bien especificar explícitamente el búfer a utilizar cuando se crea el objeto **Formatter**. También es posible enviar la salida formateada con un objeto **Formatter** a un archivo.

La clase **Formatter** define diversos constructores, los cuales nos permiten construir objetos **Formatter** en una gran variedad de formas. Éstos son algunos ejemplos:

Formatter( )

Formatter(Appendable *b)*

Formatter(Appendable *b*, Locale *loc*)

Formatter(String *nombre*)
  throws FileNotFoundException

Formatter(String *nombre*, String *charset*)
  throws FileNotFoundException, UnsupportedEncodingException

Formatter(File *f*)
  throws FileNotFoundException

Formatter(OutputStream *os*)

Aquí, *b* especifica el búfer para la salida con formato. Si *b* es null, el objeto **Formatter** automáticamente asigna espacio para un objeto **StringBuilder** que almacena la salida formateada. El parámetro *loc* especifica la localidad. Si no se especifica *localidad* se utiliza la localidad por omisión. El parámetro *nombre* especifica el nombre del archivo que recibirá la salida formateada. El parámetro *charset* especifica el conjunto de caracteres. Si no se especifica un conjunto de caracteres, se utiliza el conjunto de caracteres por omisión. El parámetro *f* especifica una referencia a un archivo abierto que recibirá la salida. El parámetro *os* especifica una referencia a un flujo de salida que recibirá el resultado. Cuando se utiliza un archivo, la salida se escribe en él.

Quizá el constructor que más se utiliza sea el primero, el constructor sin parámetros. Este constructor automáticamente utiliza la localidad por omisión y asigna espacio para un objeto **StringBuilder** que almacenará la salida.

Métodos de la clase Formatter

La clase **Formatter** define los métodos mostrados en la Tabla 18-11.

Principios de formato

Después de crear un objeto **Formatter**, se puede utilizar para crear una cadena con formato. Para hacer eso, se utiliza el método **format( )**. La versión más comúnmente utilizada de este método es la siguiente:

Formatter format(String *f*, Object … *args*)

Método	Descripción
void close( )	Cierra el objeto **Formatter** que invoca. Esto causa que cualquier recurso utilizado por el objeto sea liberado. Después de que el objeto **Formatter** ha sido cerrado, no puede ser reutilizado. Intentar utilizar un objeto **Formatter** cerrado genera una excepción de tipo **FormatterClosedException**.
void flush( )	Envía el contenido del búfer a la salida. Esto causa que la información localizada en el búfer sea escrita en el destino especificado. Esto aplica principalmente para objetos **Formatter** conectados con un archivo.
Formatter format(String *f*, Object … *args*)	Da formato al argumento dado en *args* según lo especifica el argumento *f*. Devuelve al objeto que invoca.
Formatter format(Locale *loc*, String *f*, Object … *args*)	Da formato al argumento dado en *args* según lo especifica el argumento *f*. La localidad especificada en *loc* es utilizada para el formato. Devuelve al objeto que invoca.
IOException ioException( )	Si el objeto subyacente que es el destino para la salida genera una excepción **IOException**, entonces esta excepción es devuelta. En caso contrario se devuelve null.
Locale locale( )	Devuelve la localidad del objeto que invoca.
Appendable out( )	Devuelve una referencia al objeto subyacente que es el destino para la salida.
String toString( )	Devuelve una **cadena** que contiene la salida formateada.

TABLA 18-11 Los métodos definidos por la clase **Formatter**

La cadena *f* consiste en dos tipos de elementos. El primer tipo se compone de caracteres que son simplemente copiados en el búfer de salida. El segundo tipo contiene *especificadores de formato*, los cuales definen la forma en que argumentos subsecuentes serán mostrados.

En su forma más simple, un especificador de formato comienza con un signo de porcentaje seguido de un *especificador de conversión*. Todos los especificadores de conversión consisten en un solo carácter. Por ejemplo, el especificador para datos de punto flotante es **%f**. En general, debe existir el mismo número de argumentos que de especificadores de formato. Los especificadores y los argumentos se corresponden en orden de izquierda a derecha. Por ejemplo considere el siguiente fragmento de código:

```
Formatter fmt = new Formatter();
fmt.format("Dar formato %s es fácil %d %f", "con Java", 10, 98.6);
```

Esta secuencia crea un objeto **Formatter** que contiene la siguiente cadena:

```
Dar formato con Java es fácil 10 98.600000
```

En este ejemplo, los especificadores de formato **%s**, **%d** y **%f**, son remplazados con los argumentos que siguen a la cadena de formato. Así, **%s** es remplazada por "con Java", **%d** es remplazada por 10, y **%f** es remplazado por 98.6. El resto de los caracteres se copian tal cual. Como se puede imaginar, el especificador de formato **%s** especifica una cadena, y **%d** especifica un valor entero. Como se mencionó antes, **%f** especifica un valor de punto flotante.

TABLA 18-12 Especificadores de formato

Especificador de Formato	Conversión aplicada
%a %A	Punto flotante hexadecimal
%b %B	Boolean
%c	Carácter
%d	Entero en formato decimal
%h %H	Código de dispersión del argumento
%e %E	Notación científica
%f	Punto decimal
%g %G	Utiliza **%e** o **%f**, cualquiera que sea más corto
%o	Entero en formato octal
%n	Inserta un carácter de salto de línea
%s %S	Cadena
%t %T	Hora y fecha
%x %X	Entero en formato hexadecimal
%%	Inserta un signo de %

El método **format( )** acepta una gran variedad de especificadores de formato, los cuales se muestran en la Tabla 18-12. Note que muchos especificadores tienen ambas formas, la letra en mayúscula y en minúscula. Cuando un especificador con la letra en mayúscula es utilizado, las letras se muestran en mayúsculas. Fuera de eso, el carácter en mayúsculas y en minúsculas realiza la misma conversión. Es importante entender que Java revisa los tipos de cada especificador de formato y de su argumento correspondiente. Si el argumento no corresponde con el especificador, se genera una excepción de tipo **IllegalFormatException**.

Una vez que se ha formateado una cadena, se puede obtener la cadena formateada llamando al método **toString( )**. Por ejemplo, continuando con el ejemplo anterior, la siguiente sentencia obtiene la cadena formateada contenida en el objeto llamado fmt:

```
String srt = fmt.toString( );
```

Por su puesto, si deseamos simplemente mostrar la cadena formateada, no existe razón para asignarla antes a un objeto **String**. Cuando un objeto **Formatter** se pasa como argumento al método **println( )**, por ejemplo, su método **toString( )** se llama de manera automática.

A continuación se muestra un programa pequeño que coloca juntas todas las piezas y muestra como crear y visualizar una cadena con formato:

```
// Un ejemplo simple que utiliza objetos Formatter
import java.util.*;

class FormatDemo{
  public static void main(String args[]) {
    Formatter fmt = new Formatter( );

    fmt.format("Dar formato %s es fácil %d %f", "con Java", 10, 98.6);

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

Un comentario adicional: es posible obtener una referencia al búfer subyacente de salida llamando al método **out( )**. Éste devuelve una referencia a un objeto de tipo **Appendable**.

Ahora que conocemos el mecanismo general utilizado para crear una cadena con formato, el resto de esta sección discute a detalle cada conversión. Además se describen diferentes opciones de formato, como son justificación, tamaño mínimo de un campo y precisión.

Formato de cadenas y caracteres

Para dar formato a un carácter individual, se utiliza **%c**. Esto hace que el carácter al que le corresponde el formato sea impreso sin modificación alguna. Para dar formato a una cadena, se utiliza **%s**.

Formato de números

Para dar formato a un número entero en su representación decimal, se utiliza **%d**. Para dar formato a un número de punto flotante en su representación decimal, se utiliza **%f**. Para dar formato a un número de punto flotante en su representación científica, se utiliza **%e**. Los números representados en notación científica toman la siguiente forma general:

x.dddddde+/-yy

El especificador de formato **%g** causa que el objeto **Formatter** utilice ya sea **%f** o **%e**, el que resulte más corto. El siguiente programa muestra los efectos del especificador de formato **%g**.

```
// Ejemplo del especificador de formato %g
import java.util.*;

class FormatDemo2{
  public static void main(String args[]) {
    Formatter fmt = new Formatter();

    for (double i = 1000; i < 1.0e+10; i *=100) {
      fmt.format("%g", i);
      System.out.println(fmt);
    }
  }
}
```

El programa produce la siguiente salida:

```
1000.000000
1000.000000 1000000.000000
1000.000000 1000000.000000 1.000000e+07
1000.000000 1000000.000000 1.000000e+07 1.000000e+09
```

Es posible mostrar números enteros en formato octal o hexadecimal utilizando los especificadores **%o** y **%x** respectivamente. Por ejemplo el siguiente código:

```
fmt.format("Hexadecimal: %x, Octal: %o", 196, 196);
```

produce esta salida:

```
Hexadecimal: c4, Octal: 304
```

Podemos visualizar valores de punto flotante en formato hexadecimal utilizando el especificador **%a**. El formato producido por **%a** parece un poco extraño a primera vista. Esto es

debido a que su representación utiliza una forma similar a la notación científica, la cual consiste en una base y un exponente, ambos en hexadecimal. Esta es la forma general:

0x1.*sig*p*exp*

Aquí, *sig* contiene la porción fraccional de la base y *exp* contiene el exponente. La letra **p** indica el inicio del exponente. Por ejemplo, la siguiente sentencia:

```
fmt.format("%a", 123.123 );
```

produce esta salida:

```
0x1.ec7df3b645a1dp6
```

Formato de horas y fechas

Uno de los especificadores de conversión más poderosos es **%t**. Este permite formatear información de horas y fechas. El especificador **%t** trabaja de manera ligeramente diferente a los otros debido a que requiere el uso de un sufijo para describir la porción y formato preciso de la hora o fecha deseadas. Los sufijos se muestran en la Tabla 18-13. Por ejemplo, para desplegar minutos, se utiliza **%tM**, donde **M** indica minutos desplegados utilizando dos caracteres de texto. El argumento correspondiente al especificador **%t** debe ser de tipo **Calendar**, **Date**, **Long** o **long**.

El siguiente programa muestra varios de los formatos

```
// Ejemplo del formatos para fecha y hora
import java.util.*;

class TimeDateFormat{
  public static void main(String args[]) {
    Formatter fmt = new Formatter( );
    Calendar cal = Calendar.getInstance( );

    // Visualiza un formato de estándar a 12 horas
    fmt.format("%tr", cal);
    System.out.println(fmt);

    // Visualiza la información completa de hora y fecha
    fmt = new Formatter( );
    fmt.format("%tc", cal);
    System.out.println(fmt);

    // Visualiza sólo la información de horas y minutos
    fmt = new Formatter( );
    fmt.format("%tl:%tM", cal, cal);
    System.out.println(fmt);

    // Visualiza el nombre y número de mes
    fmt = new Formatter( );
    fmt.format("%tB %tb %tm", cal, cal, cal);
    System.out.println(fmt);
  }
}
```

La salida sería como la siguiente:

```
09:17:15 AM
Thu Jan 01 09:17:15 CST 2009
9:17
January Jan 01
```

TABLA 18-13
Sufijos de formato para hora y fecha

Sufijo	Se remplaza con
a	Nombre del día de la semana abreviado
A	Nombre del día de la semana completo
b	Nombre del mes abreviado
B	Nombre del mes completo
c	Cadena estándar de fecha y hora formateada como: *día mes fecha hh:mm:ss zona año*
C	Primeros dos dígitos del año
d	Día del mes como número decimal (01 a 31)
D	mes / día / año
e	Día del mes como número decimal (1 a 31)
F	año – mes – día
h	Nombre del mes abreviado
H	Hora (00 a 23)
I	Hora (01 a 12)
j	Día del año como número decimal (0001 a 366)
k	Hora (0 a 23)
l	Hora (1 a 12)
L	Milisegundos (000 a 999)
m	Mes como número decimal (01 a 13)
M	Minuto como número decimal (00 a 59)
N	Nanosegundo (000000000 a 999999999)
p	Equivalente local de AM o PM en minúsculas
Q	Milisegundos desde 1/1/1970
r	*hh:mm:ss* (formato de 12 horas)
R	*hh:mm* (formato de 24 horas)
S	Segundos (00 a 60)
s	Segundos desde 1/1/1970 UTC
T	*hh:mm:ss* (formato de 24 horas)
y	Año como número decimal sin centenas (00 a 99)

TABLA 18-13 Sufijos de formato para hora y fecha (*continuación*)

Sufijo	Se reemplaza con
Y	Año como número decimal con centenas (0001 a 9999)
z	Desplazamiento desde UTC
Z	Nombre de la zona horaria

Los especificadores %n y %%

Los especificadores de formato **%n** y **%%** difieren de los otros debido a que no se corresponden con un argumento. En lugar de ello, representan simplemente secuencias de escape que insertan un carácter en la secuencia de salida. El especificador **%n** inserta una nueva línea. El especificador **%%** inserta un signo de porcentaje. Ninguno de esos caracteres puede ser ingresado directamente en la cadena formateada. Por supuesto, es posible utilizar la secuencia de escape convencional **\n** para incrustar un carácter de salto de línea.

El siguiente ejemplo muestra el uso de los especificadores de formato **%n** y **%%**

```
// Ejemplo de los especificadores de formato %n y %%
import java.util.*;

class FormatDemo3{
  public static void main(String args[]) {
    Formatter fmt = new Formatter();

    fmt.format("Copiando archivo%nTransferencia %d%% completa", 88);
    System.out.println(fmt);
  }
}
```

El programa despliega la siguiente salida:

```
Copiando archivo
Transferencia 88% completa
```

Especificación del tamaño mínimo de un campo

Un valor entero colocado en medio de signos de **%** acompañado de un código de conversión de formato actúa como un especificador de *tamaño mínimo de un campo*. Esto rellena la salida con espacios para asegurar que alcance cierto tamaño mínimo. Si la cadena o número es mayor que el mínimo establecido, se mostrará completa. Por omisión el relleno se hace con espacios en blanco. Si el programador necesita rellenar con ceros, entonces se coloca un cero antes del especificador de tamaño. Por ejemplo, **%05d** rellenará un número de menos de cinco dígitos con ceros de forma que su tamaño total sea cinco. El especificador de tamaño del campo puede ser utilizado con todos los especificadores de formato excepto **%n**.

El siguiente programa es un ejemplo del especificador de tamaño mínimo para un campo, aplicado a una conversión con el espeficificador **%f**.

```
// Ejemplo del especificado de tamaño mínimo
import java.util.*;

class FormatDemo4{
  public static void main(String args[]) {
    Formatter fmt = new Formatter();
```

```
    fmt.format("|%f|%n|%12f|%n|%012f|",
               10.12345, 10.12345, 10.12345);

    System.out.println(fmt);

  }
}
```

El programa anterior produce la siguiente salida:

```
|10.123450|
|  10.123450|
|00010.123450|
```

La primera línea despliega el número 10.12345 con su tamaño por omisión. La segunda línea despliega el valor en un tamaño de campo igual a 12 caracteres. La tercera línea despliega el valor en un tamaño de campo igual a 12 caracteres, rellenos con ceros iniciales.

El especificador de tamaño mínimo de un campo se utiliza a menudo para producir tablas en las cuales las columnas estén alineadas. Por ejemplo el siguiente programa produce una tabla de cuadrados y cubos para los números entre 1 y 10.

```
// Crea una tabla de cuadrados y cubos.
import java.util.*;

class FieldWidthDemo{
  public static void main(String args[]) {
    Formatter fmt;

    for (int i = 1; i <=10; i++) {
      fmt = new Formatter();

      fmt.format("%4d %4d %4d", i, i*i, i*i*i );
      System.out.println(fmt);
    }

  }
}
```

La salida generada es la siguiente:

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

Especificación de precisión

Un *especificador de precisión* puede ser aplicado a los especificadores de formato **%f**, **%e**, **%g** y **%s**. Se coloca a continuación del especificador de tamaño mínimo de campo (si existe uno) y consiste en un punto seguido por un número entero. Su significado exacto depende del tipo de dato al cual se aplica.

Cuando se aplica el especificador de precisión a un dato de punto flotante utilizando los especificadores **%f** o **%e**, éste determina el número de lugares decimales a mostrar. Por ejemplo, **%10.4f** muestra un número con al menos 10 caracteres de tamaño con cuatro decimales. Cuando se utiliza **%g**, la precisión determina el número de dígitos significativos. La precisión por omisión es 6.

Aplicado a cadenas, el especificador de precisión define la longitud máxima del campo. Por ejemplo, **%5.7s** muestra una cadena con al menos cinco y no más de siete caracteres de longitud. Si la cadena es más larga que el tamaño máximo del campo, los últimos caracteres serán truncados.

El siguiente programa ilustra el uso del especificador de precisión:

```
// Ejemplo del especificador de precisión
import java.util.*;

class PresicionDemo{
  public static void main(String args[]) {
    Formatter fmt = new Formatter();

    // Formato con 4 lugares decimales
    fmt.format("%.4f", 123.1234567);
    System.out.println(fmt);

    // Formato con 2 lugares decimales en un campo de 16 caracteres
    fmt = new Formatter();
    fmt.format("%16.2e", 123.1234567);
    System.out.println(fmt);

    // Mostrar como máximo 15 caracteres en una cadena
    fmt = new Formatter();
    fmt.format("%15s", "Dar formato con Java es fácil.");
    System.out.println(fmt);
  }
}
```

El programa produce la siguiente salida:

```
    123.1235
            1.23e+02
    Dar formato con
```

Uso de las banderas de formato

La clase **Formatter** reconoce un conjunto de banderas de formato que permiten controlar varios aspectos de la conversión. Todas las banderas de formato son caracteres simples, una bandera de formato se coloca a continuación del carácter % en una especificación de formato. Las banderas de formato se muestran a continuación:

Bandera	Efecto
–	Justificación a la izquierda
#	Alternar el formato de conversión
0	La salida se rellena con ceros en lugar de espacios en blanco
espacio	Una salida numérica con valor positivo aparece precedida con un espacio en blanco

Bandera	Efecto
+	Una salida numérica con valor positivo aparece precedida con un signo de +
,	Los valores numéricos incluyen separadores de grupos
(	Los valores numéricos negativos se encierran entre paréntesis.

No todas las banderas aplican para todos los especificadores de formato. Las siguientes secciones explican cada una de las banderas con detalle.

Justificado del texto de salida

Por omisión, se realiza una justificación a la derecha. Esto es, si el tamaño del campo es mayor que el dato a mostrar, el dato será colocado al lado derecho del campo. Es posible obligar a que la salida se justifique a la izquierda colocando un signo de menos directamente después del carácter %. Por ejemplo, **%-10.2f** justifica a la izquierda un número de punto flotante con dos lugares decimales en un campo de 10 caracteres de largo. Por ejemplo, considere el siguiente programa:

```
// Ejemplo de justificación a la izquierda
import java.util.*;

class LeftJustify{
  public static void main(String args[]) {
    Formatter fmt = new Formatter( );

    // Justificación a la derecha por omisión
    fmt.format("|%10.2f|", 123.123);
    System.out.println(fmt);

    // Ahora, justificación a la izquierda
    fmt = new Formatter( );

    fmt.format("|%-10.2f|", 123.123);
    System.out.println(fmt);
  }
}
```

El programa produce la siguiente salida:

```
|    123.12|
|123.12    |
```

Como se puede ver, la segunda línea está justificada a la izquierda en un campo de 10 caracteres.

Las banderas de espacio, +, 0 y (

Para que un signo + sea mostrado antes de un valor numérico positivo, se añade la bandera de +. Por ejemplo,

```
fmt.format("%+d", 100);
```

Crea esta cadena:

```
+100
```

Cuando se crean columnas de números, algunas veces es útil mostrar un espacio antes de un valor positivo para que los valores positivos y negativos se alineen. Para hacer esto, se puede añadir la bandera de espacio. Por ejemplo:

```
// Ejemplo de especificadores de formato para espacio
import java.util.*;

class FormatDemo5{
  public static void main(String args[]) {
    Formatter fmt = new Formatter();

    fmt.format("% d", -100);
    System.out.println(fmt);

    fmt = new Formatter();
    fmt.format("% d", 100);
    System.out.println(fmt);

    fmt = new Formatter();
    fmt.format("% d", -200);
    System.out.println(fmt);

    fmt = new Formatter();
    fmt.format("% d", 200);
    System.out.println(fmt);
  }
}
```

La salida del programa anterior es:

```
-100
 100
-200
 200
```

Nótese que los valores positivos tienen un espacio en blanco precediéndolos, lo cual causa que las columnas de dígitos aparezcan alineadas.

Para mostrar los números negativos entre paréntesis, en lugar de mostrarlos con un signo de menos precediéndolos, utilice la bandera **(**. Por ejemplo:

```
fmt.format("%(d", -100);
```

crea la cadena:

(100)

La bandera 0 provoca que la cadena resultante sea rellenada con ceros en lugar de espacios en blanco.

La bandera del signo coma

Cuando se visualizan números grandes, a menudo es útil añadirles separadores de grupos, para lo cual se emplea comúnmente al símbolo coma. Por ejemplo, el valor 1234567 se lee con mayor facilidad cuando se formatea como 1,234,567. Para añadir especificadores de grupo, se utiliza la bandera de signo coma (,). Por ejemplo,

```
fmt.format("%,.2f", 4356783497.34);
```

crea la cadena:

```
4,356,783,497.34
```

La bandera de

La bandera **#** puede ser aplicada a los especificadores **%o**, **%x**, **%e** y **%f**. Para **%e** y **%f**, la bandera **#** asegura que habrá un punto decimal incluso si no están presentes dígitos decimales. Si se precede el especificador de formato **%x** con un **#**, el número hexadecimal será impreso con el prefijo **0x**. Precediendo **%o** con el especificador **#** causará que el número se imprima con un cero inicial.

La opción mayúsculas

Como se mencionó antes, varios especificadores de formato tienen versiones en mayúsculas que ocasionan que la conversión utilice letras mayúsculas cuando sea apropiado. La siguiente tabla describe los efectos obtenidos.

Especificador	Efecto
%A	Causa que los dígitos hexadecimales de la *a* a la *f* sean mostrados en mayúsculas como *A* a *F* respectivamente. Además, el prefijo **0x** será mostrado como **0X**, y la letra **p** será desplegada como **P**.
%B	Coloca en mayúsculas los valores **true** y **false**.
%E	Causa que el símbolo *e* que representa el inicio del exponente sea mostrado en mayúsculas.
%G	Causa que el símbolo *e* que representa el inicio del exponente sea mostrado en mayúsculas.
%H	Causa que los dígitos hexadecimales de la *a* a la *f* sean mostrados en mayúsculas como A a F.
%S	Cambia a mayúsculas la cadena a la cual se aplica.
%T	Causa que todos los caracteres alfabéticos sean mostrados en mayúsculas.
%X	Causa que los dígitos hexadecimales de la *a* a la *f* sean mostrados en mayúsculas como *A* a *F*. Además, si está presente el prefijo **0x** será mostrado como **0X**.

Por ejemplo, la siguiente sentencia:

```
fmt.format("%X", 250);
```

crea la siguiente cadena:

```
FA
```

Esta línea:

```
fmt.format("%E", 123.1234);
```

crea la cadena:

```
1.231234E+02
```

Uso de índices de argumento

La clase **Formatter** incluye una característica muy útil que permite especificar a cuál argumento será aplicado un especificador de formato. Normalmente, los especificadores de formato y los argumentos son acoplados en orden de izquierda a derecha. Esto es, el primer especificador de formato se aplica al primer argumento, el segundo especificador de formato al segundo argumento y así sucesivamente. Sin embargo, utilizando *índices de argumento*, podemos controlar explícitamente a que argumento se aplica un especificador de formato.

Un índice de argumento se coloca inmediatamente después del símbolo **%** del especificador de formato. Éste tiene la siguiente forma:

n$

Donde *n* es el índice del argumento deseado, comenzando con 1. Por ejemplo, considere la siguiente línea:

```
fmt.format("%3$d %1$d %2$d", 10, 20, 30);
```

la cual produce la siguiente salida:

```
30 10 20
```

En este ejemplo, el primer especificador de formato corresponde a 30, el segundo corresponde a 10 y el tercero a 20. Así, los argumentos se utilizan en un orden diferente al orden de izquierda a derecha.

Una ventaja de los índices de argumento es que nos permiten reutilizar un argumento sin tener que especificarlo dos veces. Por ejemplo, considere esta línea:

```
fmt.format("%d en hexadecimal es %1$x", 255);
```

La línea anterior produce esta salida:

```
255 en hexadecimal es ff
```

Como podemos ver, el argumento 255 se utiliza por ambos especificadores de formato.

Existe una forma corta, más cómoda, llamada *índices relativos* que permite reutilizar el argumento utilizado por el especificador de formato precedente. Esto se hace simplemente colocando un signo < en el índice del argumento. Por ejemplo, la siguiente llamada al método **format( )** produce los mismos resultados que el ejemplo anterior:

```
fmt.format("%d en hexadecimal es %<x", 255);
```

Los índices relativos son especialmente útiles cuando se crean formatos de hora y fecha personalizados. Considere el siguientes ejemplo:

```
// Uso de índices relativos para simplificar
// la creación de formatos personalizados para fecha y hora
import java.util.*;

class FormatDemo6{
  public static void main(String args[]) {
    Formatter fmt = new Formatter();
    Calendar cal = Calendar.getInstance();

    fmt.format("Hoy es el día %te de %<tB, %<tY", cal);
```

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

Este es un ejemplo de la salida del programa anterior:

```
Hoy es el día 1 de Jan, 2007
```

Con el uso de los índices relativos, el argumento **cal** sólo necesita ser colocado una vez, en lugar de tres.

El método printf( )

Aunque técnicamente no existe nada malo en utilizar **Formatter** directamente (como se ha hecho en los ejemplos anteriores) cuando se crean cadenas que serán mostradas en pantalla, existe una alternativa más cómoda: el método **printf( ).** El método **printf( )** automáticamente utiliza la clase **Formatter** para crear una cadena con formato. Este método luego muestra la cadena en **System**.**out, System.out** representa a la consola por omisión. El método **printf( )** está definido por las clases **PrintStream** y **PrintWriter**. El método **printf( )** se describe en el Capítulo 19.

Scanner

La clase **Scanner** es el complemento de la clase **Formatter**. Añadida en el JDK 5, la clase **Scanner** lee una entrada formateada y la convierte en su forma binaria. Aunque siempre ha sido posible leer entradas formateadas, esto requería mayor esfuerzo de lo que muchos programadores consideraban apropiado. Debido a la adición de la clase **Scanner**, hoy día es más fácil leer todo tipo de valores numéricos, cadenas, y otros tipos de datos, tanto desde un archivo en disco, el teclado, o cualquier otra fuente.

La clase **Scanner** puede ser utilizada para leer desde consola, un archivo, una cadena, o cualquier otra fuente que implemente la interfaz **Readable** o **ReadableByteChannel**. Por ejemplo, se puede utilizar **Scanner** para leer un número del teclado y asignar este valor a una variable. Como se verá más adelante, dado su poder, **Scanner** es sorprendentemente fácil de utilizar.

Constructores de la clase Scanner

La clase **Scanner** define los constructores mostrados en la Tabla 18-14. En general, un objeto **Scanner** puede ser creado por un objeto **String**, **InputStream**, **File** o cualquier otro objeto que implemente la interfaz **Readable** o la interfaz **ReadableByteChannel**. A continuación algunos ejemplos.

La siguiente secuencia crea un objeto **Scanner** que lee el archivo **Test.txt**:

```
FileReader fin = new FileReader("Test.txt");
Scanner src = new Scanner (fin);
```

Esto funciona debido a que **FileReader** implementa la interfaz **Readable**. Así, la llamada al constructor se resuelve como **Scanner(Readable)**.

Método	Descripción
Scanner(File *f*) throws FileNotFoundException	Crea un objeto **Scanner** que utiliza el archivo especificado por *f* como fuente de entrada.
Scanner(File *f*, String *charset*) throws FileNotFoundException	Crea un objeto **Scanner** que utiliza el archivo especificado por *f* como fuente de entrada con la codificación especificada por *charset.*
Scanner(InputStream *f*)	Crea un objeto **Scanner** que utiliza el flujo especificado por *f* como fuente de entrada.
Scanner(InputStream *f*, String *charset*)	Crea un objeto **Scanner** que utiliza el flujo especificado por *f* como fuente de entrada con la codificación especificada por *charset.*
Scanner(Readable *f*)	Crea un objeto **Scanner** que utiliza el objeto tipo **Readable** especificado por *f* como fuente de entrada.
Scanner(ReadableByteChannel *f*)	Crea un objeto **Scanner** que utiliza el objeto tipo **ReadableByteChannel** especificado por *f* como fuente de entrada.
Scanner(ReadableByteChannel *f*, String *charset*)	Crea un objeto **Scanner** que utiliza el objeto tipo **ReadableByteChannel** especificado por *f* como fuente de entrada con la codificación especificada por *charset*
Scanner(String *f*)	Crea un objeto **Scanner** que utiliza la cadena especificada por *f* como fuente de entrada.

TABLA 18-14 Los constructores de la clase **Scanner**

Las siguientes líneas crean un objeto **Scanner** que lee de la entrada estándar, es decir, desde el teclado:

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

Esto funciona debido a que **System.in** es un objeto de tipo **InputStream**. Así, la llamada al constructor se realiza como **Scanner(InputStream)**.

La siguiente secuencia crea un objeto **Scanner** que lee datos desde una cadena.

```
String cadena = "10 99.88 esto es fácil";
Scanner conin = new Scanner(cadena);
```

Funcionamiento de Scanner

Una vez que hemos creado un objeto **Scanner**, es simple utilizarlo para leer una entrada formateada. En general, un objeto **Scanner** lee *tokens* de la fuente subyacente especificada por el programador cuando el objeto **Scanner** es creado. Un token es una porción del texto de entrada que está delineada por un conjunto de delimitadores, por omisión, delimitada por espacios en blanco. Un token es leído cuando éste corresponde con una *expresión regular*, la expresión regular define el formato de los datos. Aunque **Scanner** permite definir el tipo específico de expresión con la que la siguiente operación de entrada deberá coincidir, esta clase incluye muchos patrones

predefinidos, los cuales corresponden a los tipos primitivos, como **int** y **double** y cadenas. Por ello, con frecuencia el programador no necesita especificar un patrón.

En general, para utilizar **Scanner**, sigamos el siguiente procedimiento:

1. Se determina si un tipo específico de entrada está disponible, llamando a uno de los métodos **hasNextX** de la clase **Scanner**, donde *X* es el tipo de dato deseado.
2. Si existe una entrada disponible, se lee llamando a uno de los métodos **nextX** de la clase **Scanner**.
3. Se repite el proceso hasta que los datos de entrada terminan.

Como se indicó anteriormente, la clase **Scanner** define dos conjuntos de métodos que nos permiten leer la entrada. El primer conjunto está formado por los métodos **hasNextX**, los cuales se muestran en la Tabla 18-15. Estos métodos determinan si un tipo de entrada específico está disponible. Por ejemplo, la llamada al método **hasNextInt( )** devuelve **true** solo si el siguiente token disponible es un entero. Si el dato deseado está disponible, se puede leer llamando a uno de los métodos **nextX** de la clase **Scanner**, los cuales se muestran en la Tabla 18-16. Por ejemplo, para leer el siguiente número entero, se llama al método **nextInt( )**. La siguiente secuencia muestra cómo leer una lista de enteros desde el teclado.

```
Scanner conin = new Scanner(System.in);
int i;

// Leer una lista de enteros
while(conin.hasNextInt()) {
  i = conin.nextInt();
  // …
}
```

Método	Descripción
boolean hasNext( )	Devuelve **true** si otro token de cualquier tipo está disponible para ser leído. Devuelve **false** en caso contrario.
boolean hasNext(Pattern *p*)	Devuelve **true** si un token que corresponde con el patrón especificado en *p* está disponible para ser leído. Devuelve **false** en caso contrario.
boolean hasNext(String *p*)	Devuelve **true** si un token que corresponde con el patrón especificado en *p* está disponible para ser leído. Devuelve **false** en caso contrario.
boolean hasNextBigDecimal( )	Devuelve **true** si un valor que puede ser almacenado en un objeto **BigDecimal** está disponible para ser leído. Devuelve **false** en caso contrario.
boolean hasNextBigInteger( )	Devuelve **true** si un valor que puede ser almacenado en un objeto **BigInteger** está disponible para ser leído. Devuelve **false** en caso contrario. Utiliza la base matemática por omisión (a menos que sea modificada la base por omisión es 10).
boolean hasNextBigInteger(int *base*)	Devuelve **true** si un valor en la *base* especificada y que puede ser almacenado en un objeto **BigInteger** está disponible para ser leído. Devuelve **false** en caso contrario.

TABLA 18-15 Métodos **hasNext** de la clase **Scanner**

Método	Descripción
boolean hasNextBoolean( )	Devuelve **true** si un valor **boleano** está disponible para ser leído. Devuelve **false** en caso contrario.
boolean hasNextByte( )	Devuelve **true** si un valor de tipo **byte** está disponible para ser leído. Devuelve **false** en caso contrario. Utiliza la base matemática por omisión (a menos que sea modificada la base por omisión es 10).
boolean hasNextByte(int *base*)	Devuelve **true** si un valor de tipo **byte** en la *base* especificada está disponible para ser leído. Devuelve **false** en caso contrario.
boolean hasNextDouble( )	Devuelve **true** si un valor de tipo **double** está disponible para ser leído. Devuelve **false** en caso contrario.
boolean hasNextDouble( )	Devuelve **true** si un valor de tipo **float** está disponible para ser leído. Devuelve **false** en caso contrario.
boolean hasNextInt( )	Devuelve **true** si un valor de tipo **int** está disponible para ser leído. Devuelve **false** en caso contrario. Utiliza la base matemática por omisión (a menos que sea modificada la base por omisión es 10).
boolean hasNextInt(int *base*)	Devuelve **true** si un valor de tipo int en la *base* especificada está disponible para ser leído. Devuelve **false** en caso contrario.
boolean hasNextLine( )	Devuelve **true** si una línea está disponible para ser leída.
boolean hasNextLong( )	Devuelve **true** si un valor de tipo long está disponible para ser leído. Devuelve **false** en caso contrario. Utiliza la base matemática por omisión (a menos que sea modificada la base por omisión es 10).
boolean hasNextLong(int *base*)	Devuelve **true** si un valor de tipo **long** en la *base* especificada está disponible para ser leído. Devuelve **false** en caso contrario.
boolean hasNextShort( )	Devuelve **true** si un valor de tipo **short** está disponible para ser leído. Devuelve **false** en caso contrario. Utiliza la base matemática por omisión (a menos que sea modificada la base por omisión es 10).
boolean hasNextShort(int *base*)	Devuelve **true** si un valor de tipo **short** en la *base* especificada está disponible para ser leído. Devuelve **false** en caso contrario.

TABLA 18-15 Métodos **hasNext** de la clase **Scanner** (*continuación*)

Método	Descripción
String next( )	Devuelve el siguiente token de cualquier tipo, tomándolo de la fuente de entrada.
String next(Pattern *p*)	Devuelve el siguiente token que coincide con el patrón dado en *p*, tomándolo de la fuente de entrada.
String next(String *p*)	Devuelve el siguiente token que coincide con el patrón dado en *p*, tomándolo de la fuente de entrada.
BigDecimal nextBigDecimal( )	Devuelve el siguiente token como un objeto **BigDecimal.**
BigInteger nextBigInteger( )	Devuelve el siguiente token como un objeto **BigInteger**. Utiliza la base matemática por omisión (a menos que sea modificada la base por omisión es 10).

TABLA 18-16 Métodos next de la clase **Scanner**

Método	Descripción
BigInteger nextBigInteger(int *base*)	Devuelve el siguiente token, como un objeto **BigInteger**, en la *base* especificada.
boolean nextBoolean( )	Devuelve el siguiente token como un valor **boleano**.
byte nextByte( )	Devuelve el siguiente token como un valor **byte**. Utiliza la base matemática por omisión (a menos que sea modificada la base por omisión es 10).
byte nextByte (int *base*)	Devuelve el siguiente token, como un valor de tipo **byte**, en la *base* especificada.
double nextDouble( )	Devuelve el siguiente token, como un valor de tipo **double**.
double nextFloat( )	Devuelve el siguiente token, como un valor de tipo **float**.
int nextInt( )	Devuelve el siguiente token como un valor de tipo **int**. Utiliza la base matemática por omisión (a menos que sea modificada la base por omisión es 10).
int nextInt(int *base*)	Devuelve el siguiente token como un valor de tipo **int** en la *base* especificada.
String nextLine( )	Devuelve la siguiente línea de la entrada como una cadena.
long nextLong( )	Devuelve el siguiente token como un valor de tipo **long**. Utiliza la base matemática por omisión (a menos que sea modificada la base por omisión es 10).
long nextLong(int *base*)	Devuelve el siguiente token como un valor de tipo long en la *base* especificada.
short nextShort( )	Devuelve el siguiente token como un valor de tipo **short**. Utiliza la base matemática por omisión (a menos que sea modificada la base por omisión es 10).
short nextshort(int *base*)	Devuelve el siguiente token como un valor de tipo **short** en la base especificada.

TABLA 18-16 Métodos next de la clase **Scanner** (*continuación*)

El ciclo **while** termina al momento en que el siguiente token no es un entero. Así, el ciclo detiene la lectura de enteros al momento en que un valor no entero es encontrado en el flujo de entrada.

Si un método **next** no puede encontrar el tipo de dato que está buscando genera una excepción del tipo **NoSuchElementException**. Por esta razón es mejor confirmar primero que el tipo de información deseada está disponible llamando a un método **hasNext** antes de llamar al método **next** correspondiente.

Ejemplos con la clase Scanner

La adición de la clase **Scanner** a Java convierte lo que antes era una tarea tediosa en una tarea sencilla. Para entender por qué, veamos algunos ejemplos. El siguiente programa calcula el promedio de una lista de números capturados desde el teclado:

```
// Uso de la clase Scanner para calcular un promedio de valores
import java.util.*;

class Promedio{
  public static void main(String args[]) {
    Scanner conin = new Scanner (System.in);
```

```
    int count = 0;
    double sum = 0.0;

    System.out.println("Escriba los números a promediar.");

    // Leer y sumar los números
    while (conin.hasNext()) {
      if (conin.hasNextDouble()) {
        sum += conin.nextDouble();
        count++;
      } else {
        String str = conin.next();
        if (str.equals("fin")) break;
        else {
          System.out.println("Error de formato de dato.");
          return;
        }
      }
    }

    System.out.println("El promedio es " + sum / count);
  }
}
```

El programa lee números desde el teclado y los suma, hasta que el usuario teclee la palabra "fin". El programa termina la lectura de datos desde el teclado e imprime el promedio de los números. Éste es un ejemplo de la ejecución del programa:

```
Escriba los números a promediar.
1.2
2
3.4
4
fin
El promedio es 2.65
```

El programa lee números hasta que encuentra un token que no representa un valor **double** válido. Cuando esto ocurre, se verifica si el token en cuestión es la cadena "fin". Si lo es, el programa termina normalmente. En caso contrario despliega un mensaje de error.

Obsérvese que los números son leídos llamando al método **nextDouble( )**. Este método lee cualquier número que pueda ser convertido en un valor **double**, incluyendo enteros, como el 2, y valores de punto flotante como 3.4. Así, un número leído por el método **nextDouble( )** no necesita contener un punto decimal. Este mismo principio aplica para todos los métodos **next**. Los métodos **next** pueden leer cualquier formato de dato que represente el tipo de valor solicitado.

Una cosa que es especialmente agradable de la clase **Scanner** es que la misma técnica utilizada para leer desde una fuente puede utilizarse para leer desde cualquier otra fuente. Por ejemplo, éste es el programa anterior reconstruido para promediar una lista de números contenidos en un archivo de texto:

```
// Uso de la clase Scanner para calcular el promedio de valores
contenidos en un archivo
import java.util.*;
import java.io.*;
```

```
class PromedioArchivo{
  public static void main(String args[])
    throws IOException{

    int count = 0;
    double sum = 0.0;

    // Escribir la salida en un archivo
    FileWriter fout = new FileWriter("test.txt");
    fout.write("2 3.4 5 6 7.4 9.1 10.5 fin");
    fout.close( );

    FileReader fin = new FileReader("test.txt");

    Scanner src = new Scanner (fin);

    // Leer y sumar los números
    while (src.hasNext()) {
      if (src.hasNextDouble()) {
      sum += src.nextDouble();
        count++;
      }
      else {
        String str= src.next();
        if (str.equals("fin")) break;
        else {
          System.out.println("Error de formato de dato.");
          return;
        }
      }
    }

    fin.close( );
    System.out.println("El promedio es " + sum / count);
  }
}
```

La salida generada por el programa es la siguiente:

```
El Promedio es 6.2
```

Se puede utilizar **Scanner** para leer entradas que contienen diversos tipos de datos, –aun cuando el orden de dichos datos no sea conocido de antemano. El programador puede simplemente revisar cual tipo de dato está disponible antes de leerlo. Por ejemplo, considere el siguiente programa:

```
// Uso de Scanner para leer varios tipos de datos desde un archivo
import java.util.*;
import java.io.*;

class ScanMezclado{
  public static void main(String args[])
    throws IOException{

    int i;
    double d;
    boolean b;
    String str;

    // Escribir la salida en un archivo
    FileWriter fout = new FileWriter("test.txt");
```

```
    fout.write("Probando la clase Scanner 10 12.2 uno true dos false");
    fout.close();

    FileReader fin = new FileReader("test.txt");

    Scanner src = new Scanner (fin);


    // Leer
    while (src.hasNext()) {
      if (src.hasNextInt( )) {
        i = src.nextInt();
        System.out.println("int: " + i);
      } else if (src.hasNextDouble( )) {
        d = src.nextDouble();
        System.out.println("double: " + d);
      } else if (src.hasNextBoolean()) {
        b = src.nextBoolean();
        System.out.println("boolean: " + b);
      } else {
        str = src.next( );
          System.out.println("String: " + str);
      }
    }

    fin.close( );
  }
}
```

La salida generada por el programa es la siguiente:

```
String: Probando
String: la
String: clase
String: Scanner
int: 10
double: 12.2
String: uno
boolean: true
String: dos
boolean: false
```

Cuando se leen tipos de datos mezclados, como lo hace el programa anterior, es necesario que el programador sea cuidadoso sobre el orden en el cual se llaman los métodos **next**. Por ejemplo, si el ciclo invierte el orden de las llamadas a **nextInt( )** y **nextDouble( )**, los valores numéricos serían leídos como de tipo **double**, debido a que **nextDouble( )** corresponde con cualquier cadena numérica.

Establecer los delimitadores a utilizar

La clase **Scanner** define donde inicia y termina un token con base a un conjunto de *delimitadores*. Los delimitadores por omisión son los caracteres de espacio en blanco, y éste es el conjunto de delimitadores utilizado en los ejemplos anteriores. Sin embargo, es posible cambiar los delimitadores llamando al método **useDelimiter( )**, mostrado aquí:

Scanner useDelimiter(String *p*)

Scanner useDelimiter(Pattern *p*)

Donde, *p* es una expresión regular que especifica el conjunto de delimitadores.

El siguiente programa reconstruye el programa de promedios mostrado antes de forma que ahora lea una lista de números separados por comas, y cualquier cantidad de espacios:

```
// Uso de Scanner para calcular el promedio de una lista de
// valores separados por comas
import java.util.*;
import java.io.*;

class DelimitadoresPropios{
  public static void main(String args[])
    throws IOException{

    int count = 0;
    double sum = 0.0;

    // Escribir la salida en un archivo
    FileWriter fout = new FileWriter("test.txt");

    // Ahora los valores se almacenan separador por comas
    fout.write("2, 3.4,     5, 6, 7.4,   9.1,   10.5,  fin");
    fout.close();

    FileReader fin = new FileReader("test.txt");

    Scanner src = new Scanner (fin);

    // Establecer la coma y el espacio como delimitadores
    src.useDelimiter(", *");

    // Leer y sumar los números
    while (src.hasNext()) {
      if (src.hasNextDouble()) {
        sum += src.nextDouble();
        count++;
      }
      else {
        String str= src.next();
        if (str.equals("fin")) break;
        else {
          System.out.println("Error de formato de dato.");
          return;
        }
      }
    }

    fin.close();
    System.out.println("El promedio es " + sum / count);
  }
}
```

En esta versión, los números escritos en **test.txt** están separados por comas y espacios. El uso del patrón de delimitadores", *"le dice a **Scanner** que debe buscar coincidencias con una coma y cero o mas espacios como delimitadores. La salida es la misma que antes.

Podemos obtener el patrón actual de delimitadores llamando al método **delimiter( )**, mostrado aquí:

Pattern delimiter( )

Características adicionales de la clase Scanner

La clase Scanner define muchos otros métodos adicionales a aquellos ya discutidos. Uno que es particularmente útil en algunas circunstancias es **findInLine( )**. Su forma general se muestra aquí:

String findInLine(Pattern *p*)
String findInLine(String *p*)

Este método busca el patrón especificado dentro de la siguiente línea de texto. Si el patrón es encontrado, el token correspondiente se consume y es devuelto. En caso contrario, se devuelve null. Esto funciona independientemente del conjunto de delimitadores. Este método es útil si deseamos localizar un patrón específico. Por ejemplo, el siguiente programa localiza el campo Edad en la cadena de entrada y muestra el valor asociado en pantalla:

```
// Ejemplo de findInLine()
import java.util.*;

class FindInLineDemo{
  public static void main(String args[]) {
    String instruccion = "Nombre: Javier Edad: 32 ID: 75";

    Scanner conin = new Scanner (instruccion);

    // Ubicar y desplegar la edad
    conin.findInLine("Edad:"); //encontrar Edad

    if (conin.hasNext())

      System.out.println(conin.next());
    else
      System.out.println("Error.");
  }
}
```

La salida es **28**. En el programa, el método **findInLine( )** se utiliza para encontrar una ocurrencia del patrón"Edad". Una vez encontrado, el siguiente token es leído, este siguiente token es el valor de edad.

Relacionado con **findInLine( )** se encuentra el método **findWithinHorizon( )**. Mostrado a continuación:

String findWithinHorizon(Pattern *p*, int *c*)

String findWithinHorizon(String *p*, int *c*)

Este método intenta encontrar una ocurrencia del patrón especificado dentro de los siguientes *c* caracteres. Si la búsqueda es exitosa, devuelve el patrón correspondiente. En caso contrario devuelve null. Si *c* es cero, entonces se busca dentro de la entrada hasta que se encuentra una coincidencia o se encuentra el final de la misma.

Es posible omitir un patrón utilizando el método **skip( )**, mostrado aquí:

Scanner skip(Pattern *p*)

Scanner skip(String *p*)

Si *p* coincide con una parte de la entrada, el método **skip( )** simplemente avanza más allá y devuelve una referencia al objeto que invoca. Si *p* no es encontrado, el método **skip( )** genera una excepción **NoSuchElementException**.

Otros métodos de la clase **Scanner** incluyen al método **radix( )**, el cual devuelve la base por omisión utilizada por **Scanner**; al método **useRadix( )**, el cual establece el valor de la base; el método **reset( )**, el cual restablece los valores iniciales del objeto Scanner que lo invoca; y el método **close( )**, que cierra al objeto Scanner.

Las clases ResourceBundle, ListResourceBundle y PropertyResourceBundle

El paquete **java.util** incluye tres clases que auxilian a la internacionalización de nuestros programas. La primera es la clase abstracta **ResourceBundle**. Esta clase define métodos que nos permiten administrar una colección de recursos sensibles a la localidad, como las cadenas utilizadas para etiquetar los elementos de la interfaz de usuario del programa. El programador puede definir dos o más conjuntos de cadenas traducidas que soporten diferentes lenguajes, como inglés, alemán o chino, con cada conjunto de traducciones ubicado en su propio legajo (en inglés el nombre es bundle). El programador puede después cargar el legajo adecuado a la localidad actual y utilizar las cadenas correspondientes para construir la interfaz de usuario del programa.

Los legajos de recursos se identifican por su *nombre de familia* (también denominado *nombre base*). Al nombre de familia se puede añadir un *código de lenguaje*, formado por dos caracteres en minúsculas, el cual especifica el lenguaje. En este caso, si una localidad requerida corresponde con el código de lenguaje, entonces la versión correspondiente de legajo de recursos es utilizada. Por ejemplo, un legajo de recursos con un nombre de familia **EjemploRB** puede tener una versión en alemán llamada **EjemploRB_de** y una versión en ruso llamada **EjemploRB_ru**. Note que un guión bajo separa el nombre de la familia y el código de lenguaje. En consecuencia, si la localidad es **Locale.GERMAN**, se utilizará **SampleRB_de**.

También es posible indicar una variante específica de un lenguaje que se relacione con un país específico especificando un *código de país* después del código de lenguaje. Un código de país es una cadena de dos caracteres en mayúsculas, como **AU** para Australia o **IN** para India. Un código de país también aparece precedido por un guión bajo cuando se enlaza con el nombre de un legajo de recursos.

Un legajo de recursos que sólo tiene el nombre de familia es el legajo por omisión. El legajo por omisión se utiliza cuando no se puede aplicar un legajo para un lenguaje específico.

NOTA *Los códigos de lenguaje están definidos por el estándar ISO 639 y los códigos de país por el estándar ISO 3166.*

Los métodos definidos por **ResourceBoundle** se resumen en la Tabla 18-17. Un punto importante: no se permiten claves **null** y muchos de los métodos generarán una excepción de tipo **NullPointerException** si se pasa **null** como clave. Observe la clase anidada **ResourceBoundle**.**Control.** Ésta fue añadida por Java SE 6 y es utilizada para controlar el proceso de carga de los legajos de recursos.

Existen dos subclases de **ResourceBoundle**. La primera es **PropertyResourceBundle**, la cual administra los recursos utilizando archivos de propiedades. **PropertyResourceBundle** no añade métodos propios.

Método	Descripción
static final void clearCache( )	Elimina de la memoria cache todos los legajos de recursos que fueron cargados por omisión. (Añadido por Java SE 6).
static final void clearCache(ClassLoader *cl*)	Elimina de la memoria cache todos los legajos de recursos que fueron cargados por *cl*. (Añadido por Java SE 6).
boolean containsKey(String *k*)	Devuelve **true** si *k* es una clave con el legajo de recursos que invocas (o sus ancestros) (Añadido por Java SE 6).
static final ResourceBundle getBundle(String *f*)	Carga el legajo de recursos con nombre de familia igual a *f* utilizando la localidad y el cargador de clases por omisión. Se genera una excepción de tipo **MissingResourceException** si no está disponible un legajo de recursos que coincida con el nombre de familia especificado por *f*.
static final ResourceBundle getBundle(String *f*, Locale *loc*)	Carga el legajo de recursos con nombre de familia igual a *f* utilizando la localidad especificada por *loc* y el cargador de clases por omisión. Se genera una excepción de tipo **Missing ResourceException** si no está disponible un legajo de recursos que coincida con el nombre de familia especificado por *f*.
static final ResourceBundle getBundle(String *f*, Locale *loc*, ClassLoader *cl*)	Carga el legajo de recursos con nombre de familia igual a *f* utilizando la localidad especificada por *loc* y el cargador de clases especificado por *cl*. Se genera una excepción de tipo **MissingResourceException** si no está disponible un legajo de recursos que coincida con el nombre de familia especificado por *f*.
static final ResourceBundle getBundle(String *f*, ResourceBundle.Control *c*)	Carga el legajo de recursos con nombre de familia igual a *f* utilizando la localidad y el cargador de clases por omisión. El proceso de carga queda bajo el control de *c*. Se genera una excepción de tipo **MissingResourceException** si no está disponible un legajo de recursos que coincida con el nombre de familia especificado por *f*. (Añadido por Java SE 6).
static final ResourceBundle getBundle(String *f*, Locale *loc*, ResourceBundle.Control *c*)	Carga el legajo de recursos con nombre de familia igual a *f* utilizando la localidad especificada por *loc* y el cargador de clases por omisión. El proceso de carga queda bajo el control de *c*. Se genera una excepción de tipo **MissingResourceException** si no está disponible un legajo de recursos que coincida con el nombre de familia especificado por *f* (añadido por Java SE 6).
static ResourceBundle getBundle(String *f*, Locale *loc*, ClassLoader *cl*, ResourceBundle.Control *c*)	Carga el legajo de recursos con nombre de familia igual a *f* utilizando la localidad especificada por *loc* y el cargador de clases especificado por *cl*. El proceso de carga queda bajo el control de *c*. Se genera una excepción de tipo **MissingResourceException** si no está disponible un legajo de recursos que coincida con el nombre de familia especificado por f. (Añadido por Java SE 6).
abstract Enumeration<String> getKeys( )	Devuelve las claves del legajo de recursos como una enumeración de cadenas. Cualquier clave ancestro también es considerada en el resultado.
Locale getLocale( )	Devuelve la localidad soportada por el legajo de recursos.

TABLA 18-17 Los métodos definidos por **ResourceBundle**

Método	Descripción
final Object getObject(String *k*)	Devuelve el objeto asociado con la clave dada en *k*. Genera una excepción de tipo **MissingResourceException** si *k* no está presente en el legajo de recursos.
final String getString(String *k*)	Devuelve la cadena asociada con la clave pasada vía *k*. Genera una excepción de tipo **MissingResourceException** si *k* no está en el legajo de recursos. Genera una excepción de **tipoClassCastException** si el objeto asociado con *k* no es una cadena.
final String[ ] getStringArray (String *k*)	Devuelve un arreglo de cadenas asociado con la clave dada en *k*. Genera una excepción de tipo **MissingResourceException** si *k* no está en el legajo de recursos. Genera una excepción de **tipoClassCastException** si el objeto asociado con *k* no es un arreglo de cadenas.
protected abstract Object handleGetObject(String *k*)	Devuelve el objeto asociado con la clave dada en *k*. Devuelve **null** si *k* no está en el legajo de recursos.
protected Set<String> handleKeySet( )	Devuelve las claves del legajo de recursos como un conjunto de cadenas. Las claves ancestro no se incluyen en el resultado. Tampoco las claves con valores **null**. (Añadido por Java SE 6).
Set<String> keySet( )	Devuelve las claves del legajo de recursos como un conjunto de cadenas. Cualquier clave ancestro es incluida en el resultado. (Añadido por Java SE 6).
protected void setParent(ResourceBundle *p*)	Establece *p* como el legajo ancestro para el legajo de recursos. Cuando se busca una clave, se buscará en el ancestro si la clave no es encontrada en el objeto de recursos que invoca.

TABLA 18-17 Los métodos definidos por **ResourceBundle** (*continuación*)

La segunda es la clase abstracta **ListResourceBundle**, la cual administra recursos utilizando un arreglo de pares clave/valor. La clase **ListResourceBundle** añade el método **getContents( )**, que debe ser implementado por todas sus subclases. El método tiene la forma:

```
protected abstract Object[ ][ ] getContents( )
```

Este método devuelve un arreglo de dos dimensiones que contiene pares de clave/valor que representan recursos. Las claves deben ser cadenas. Los valores regularmente también son cadenas pero pueden ser otro tipo de objetos.

A continuación tenemos un ejemplo que muestra el uso de los legajos de recursos. El legajo de recursos tiene el nombre de familia **EjemploRB**. Dos clases de legajos de recursos de esta familia son creados extendiendo **ListResourceBundle**. El primero es llamado **EjemploRB**, y es el legajo de recursos por omisión (éste trabaja con datos en idioma inglés). Veamos el ejemplo:

```
import java.util.*;

class EjemploRB extends ListResourceBundle{
  protected Object[ ][ ] resources = new Object [3][2];
```

```
    resources[0][0] = "title";
    resources[0][1] = "My Program";

    resources[1][0] = "StopText";
    resources[1][1] = "Stop";

    resources[2][0] = "StartText";
    resources[2][1] = "Start";

    return resources;
  }
}
```

El segundo legajo de recursos, mostrado a continuación, es **EjemploRB_de**. Que contiene una traducción al alemán.

```
import java.util.*;

// versión en alemán
class EjemploRB_de extends ListResourceBundle{
  protected Object[ ][ ] resources = new Object [3][2];

    resources[0][0] = "title";
    resources[0][1] = "Mein Programm";

    resources[1][0] = "StopText";
    resources[1][1] = "Anschlag";

    resources[2][0] = "StartText";
    resources[2][1] = "Anfang";

    return resources;
  }
}
```

El siguiente programa utilice los legajos de recursos definidos antes para desplegar las cadenas asociadas con cada clave tanto en el legajo por omisión (inglés) como en alemán.

```
// Ejemplo de legajos de recursos
import java.util.*;

class LRBDemo{
  public static void main(String args[]) {
    // cargar los recursos por omisión
    ResourceBundle rd = ResourceBundle.getBundle("EjemploRB");

    System.out.println("Versión en Inglés: ");
    System.out.println("Texto asociado a la clave title: " +
                       rd.getString("title"));

    System.out.println("Texto asociado a la clave StopText: " +
                       rd.getString("StopText"));

    System.out.println("Texto asociado a la clave StartText: " +
                       rd.getString("StartText"));

    // cargar los recursos en alemán
    ResourceBundle rd = ResourceBundle.getBundle("EjemploRB", Locale.GERMAN);

    System.out.println("\nVersión en Alemán: ");
```

```
        System.out.println("Texto asociado a la clave title: " +
                           rd.getString("title"));
        System.out.println("Texto asociado a la clave StopText: " +
                           rd.getString("StopText"));
        System.out.println("Texto asociado a la clave StartText: " +
                           rd.getString("StartText"));
    }
}
```

La salida del programa anterior es la siguiente:

```
Versión en Inglés:
Texto asociado a la clave title: My Program
Texto asociado a la clave StopText: Stop
Texto asociado a la clave StartText: Start

Versión en Alemán:
Texto asociado a la clave title: Mein Programm
Texto asociado a la clave StopText: Anschlag
Texto asociado a la clave StartText: Anfang
```


Otras clases e interfaces de utilería

Además de las clases mencionadas en las secciones anteriores, el paquete **java.util** incluye las siguientes clases:

EventListenerProxy	Extiende de la clase **EventListener** y acepta parámetros adicionales. Consúltese el Capítulo 22 para conocer más sobre listeners asociados a eventos.
EventObject	Es la superclase de todas las clases relacionadas con eventos. El Capítulo 22 tratará a detalle el tema de eventos.
FormattableFlags	Define banderas de formato que se utilizan con la interfaz **Formattable**.
PropertyPermission	Administra los permisos de las propiedades.
ServiceLoader	Proporciona un medio para encontrar proveedores de servicios (añadido por Java SE 6).
UUID	Encapsula y administra los Identificadores Universalmente únicos (UUID por sus siglas en inglés).

Las siguientes interfaces están también empaquetadas en **java.util**:

EventListener	Indica que una clase en un listener. Los eventos y los listeners se estudian en el Capítulo 22.
Formattable	Permite que una clase suministre un formato personalizado.

Los subpaquetes de java.util

Java define los siguientes subpaquetes de **java.util**:

- java.util.concurrent
- java.util.concurrent.atomic

- java.util.concurrent.lock
- java.util.jar
- java.util.logging
- java.util.prefs
- java.util.regex
- java.util.spi
- java.util.zip

Cada uno se describe brevemente a continuación.

Los paquetes java.util.concurrent, java.util.concurrent.atomic y java.util.concurrent.lock

El paquete **java.util.concurrent** junto con sus dos subpaquetes, **java.util.concurrent.atomic** y **java.util.concurrent.lock**, brinda soporte a la programación concurrente. Estos paquetes proporcionan una alternativa de alto rendimiento al uso de las características preconstruidas de Java para sincronización de operaciones. Este paquete se examina detalladamente en el Capítulo 26.

El paquete java.util.jar

El paquete **java.util.jar** permite leer y escribir archivos JAR (Java Archive).

El paquete java.util.logging

El paquete **java.util.logging** proporciona soporte para crear bitácoras, las cuales se emplean para registrar las acciones del programa, y para ayudar a encontrar errores en la lógica de programación.

El paquete java.util.prefs

El paquete **java.util.prefs** proporciona soporte para definir preferencias del usuario. Se utiliza comúnmente para apoyar la configuración de un programa.

El paquete java.util.regex

El paquete **java.util.regex** proporciona soporte para el uso de expresiones regulares. Se describe en detalle en el Capítulo 27.

El paquete java.util.spi

El paquete **java.util.spi** proporciona soporte para proveedores de servicios. Añadido por Java SE 6.

El paquete java.util.zip

El paquete **java.util.zip** permite leer y escribir archivos en los populares formatos ZIP y GZIP. Los flujos tanto de entrada como de salida para ZIP y GZIP están disponibles.

CAPÍTULO 19

Entrada/salida: explorando java.io

Este capítulo explora el paquete **java.io**, el cual proporciona soporte para las operaciones de E/S (entrada/salida). En el Capítulo 13 se presentó el sistema de E/S de Java. Aquí examinaremos dicho sistema con mayor detalle.

Como todos los programadores aprenden al inicio, la mayoría de los programas no pueden cumplir sus objetivos sin acceder a datos externos. Los datos se obtienen de una fuente de *entrada*. Los resultados de un programa se envían a un destino de *salida*. En Java, estas fuentes o destinos están ampliamente definidos. Por ejemplo, una conexión de red, un búfer de memoria o un archivo en disco pueden ser manipulados con las clases de E/S de Java. Aunque físicamente diferentes, estos elementos se manejan todos mediante la misma abstracción: un *flujo*. Un flujo, como se explicó en el Capítulo 13, es una entidad lógica que produce o que consume información. Un flujo se vincula a un dispositivo físico por medio del sistema de E/S de Java. Todos los flujos se comportan de la misma manera, aún cuando los dispositivos físicos con los que están vinculados difieran.

NOTA *Además de las capacidades del sistema de E/S que se discuten aquí, Java provee el paquete java.nio, el cuál se describe en el Capítulo 27.*

Las clases e interfaces de entrada/salida de Java

Las clases de E/S definidas por el paquete java.io se listan a continuación:

BufferedInputStream	FileWriter	PipedOutputStream
BufferedOutputStream	FilterInputStream	PipedReader
BufferedReader	FilterOutputStream	PipedWriter
BufferedWriter	FilterReader	PrintStream
ByteArrayInputStream	FilterWriter	PrintWriter
ByteArrayOutputStream	InputStream	PushbackinputStream
CharArrayReader	InputStreamReader	PushbackReader
CharArrayWriter	LineNumberReader	RandomAccessFile

Console	ObjectInputStream	Reader
DataInputStream	ObjectInputStream.GetField	SequenceInputStream
DataOutputStream	ObjectOutputStream	SerializablePermission
File	ObjectOutputStream.PutField	StreamTokenizer
FileDescriptor	ObjectStreamClass	StringReader
FileInputStream	ObjectStreamField	StringWriter
FileOutputStream	OutputStream	Writer
FilePermission	OutputStreamWriter	
FileReader	PipedInputStream	

Console fue agregado por Java SE 6.

El paquete **java.io** también contiene dos clases que fueron catalogadas en desuso y no se muestran en la tabla anterior: **LineNumberInputStream** y **StringBufferInputStream**. Estas clases no deberían utilizarse para escribir código nuevo.

Las siguientes interfaces están definidas en **java.io**:

Closeable	FileFilter	ObjectInputValidation
DataInput	FilenameFilter	ObjectOutput
DataOutput	Flushable	ObjectStreamConstants
Externalizable	ObjectInput	Serializable

Como se ve, hay muchas clases e interfaces en el paquete **java.io**. Éstas incluyen flujos de bytes y de caracteres, y serialización de objetos (almacenamiento y recuperación de objetos). Este capítulo examina varios de los componentes de E/S utilizados más comúnmente. La nueva clase **Console** también se examina. Comenzaremos nuestro análisis con una de las clases de E/S más distintivas: **File**.

File

Aunque la mayoría de las clases definidas por **java.io** operan sobre flujos, la clase **File** no lo hace. Esta clase trata directamente con los archivos y con el sistema de archivos. Esto es, la clase **File** no especifica cómo recuperar o almacenar información en archivos, sino que describe las propiedades de un archivo en sí mismo. Un objeto **File** se utiliza para obtener o manipular la información asociada a un archivo en disco, como los permisos, fecha, hora y rutas de directorio, así como para navegar por la jerarquía de subdirectorios.

Los archivos son una fuente y un destino primario de datos dentro de muchos programas. Aunque hay severas restricciones para su uso dentro de applets por razones de seguridad, los archivos siguen siendo un recurso central para almacenar información persistente y compartida. Un directorio en Java se trata simplemente como un **archivo** con una propiedad adicional: una lista de nombres de archivo que se pueden examinar con el método **list( ).**

Los siguientes constructores se pueden utilizar para crear objetos **File**:

File (String *rutaDirectorio*)
File (String *rutaDirectorio*, String *nomArchivo*)
File (File *objDir*, String *nomArchivo*)
File (URI *uriObj*)

Donde, *rutaDirectorio* es el nombre del directorio de un archivo, *nomArchivo* es el nombre del archivo o subdirectorio, *objDir* es un objeto **File** que especifica un directorio y *uriObj* es un objeto **URI** que describe un archivo.

El siguiente ejemplo crea tres archivos: **f1**, **f2** y **f3**. El primer objeto **File** se construye con un directorio como único argumento. El segundo incluye dos argumentos: el directorio y el nombre del archivo. El tercero incluye el directorio asignado a **f1** y un nombre de archivo; **f3** se refiere al mismo archivo que **f2**.

```
File f1 = new File ("/");
File f2 = new File ("/","autoexec.bat");
File f3 = new File (f1,"autoexec.bat");
```

NOTA *Java manipula correctamente los separadores de subdirectorios acorde a las convenciones de UNIX y de Windows. Si se utiliza una barra hacia delante (/) en una versión de Java sobre Windows, el directorio se seguirá localizando correctamente. Recuerde que si se está utilizando la convención Windows de la barra hacia atrás (\), dentro de una cadena será necesario utilizar su secuencia de escape (\\).*

La clase **File** define muchos métodos que obtienen las propiedades estándar de un objeto **File**. Por ejemplo, **getName( )** devuelve el nombre del archivo, **getParent( )** devuelve el nombre del directorio padre, y **exists( )** devuelve **true** si el archivo existe y **false** en caso contrario. La clase **File**, sin embargo, no es simétrica. Con esto queremos decir que hay muchos métodos que permiten *examinar* las propiedades de un objeto simple de tipo archivo, pero no existen las funciones correspondientes para cambiar esos atributos. El siguiente ejemplo muestra varios de los métodos de la clase **File**:

```
// Ejemplo de la clase File.
import java.io.File;

class FileDemo {
  static void p (String s) {
    System.out.println(s);
  }

  public static void main(String args[]) {
    File f1 = new File ("/java/COPYRIGHT");
    p("Nombre del archivo: " + f1.getName());
    p("Directorio: " + f1.getPath());
    p("Directorio absoluto: " + f1.getAbsolutePath());
    p("Padre: " + f1.getParent());
    p(f1.exists() ? "existe" : "no existe");
    p(f1.canWrite() ? "se puede escribir" : "no se puede escribir");
    p(f1.canRead() ? "se puede leer" : "no se puede leer");
    p((f1.isDirectory ( ) ? "" : "no" + " es un directorio"));
    p(f1.isFile () ? "es un archivo normal" : "podría ser un enlace");
    p(f1.isAbsolute() ? "es absoluto" : "no es absoluto");
```

```
    p("Última modificación: " + fl.lastModified());
    p("Tamaño del archivo: " + fl.length() + " Bytes");
  }
}
```

Al ejecutar este programa el resultado es similar al siguiente:

```
Nombre del archivo: COPYRIGHT
Directorio: /java/COPYRIGHT
Directorio absoluto: /java/COPYRIGHT
Padre: /java
existe
se puede escribir
se puede leer
no es un directorio
es un archivo normal
es absoluto
Última modificación: 812465204000
Tamaño del archivo: 695 Bytes
```

La mayoría de los métodos de **File** son auto explicativos. **isFile( )** e **isAbsolute( )** no lo son. **isFile( )** devuelve **verdadero** si se llama sobre un archivo, y **falso** si se llama sobre un directorio. También, **isFile( )** devuelve falso para algunos archivos especiales, como controladores de dispositivos y enlaces, de modo que este método se puede utilizar para asegurarse de que el archivo se comportará como tal. El método **isAbsolute( )** devuelve **verdadero** si el archivo tiene una ruta absoluta, y **falso** si su ruta es relativa.

File también incluye dos métodos de utilería. El primero es **renameTo( )**, definido como:

boolean renameTo (File *nuevoNombre*)

Donde el nombre de archivo indicado en *nuevoNombre* se convierte en el nuevo nombre del objeto **File** que invoca. Devolverá **verdadero** si la operación se lleva acabo con éxito y **falso** si no ha sido posible renombrar al archivo (por ejemplo, si se intenta renombrar un archivo con un nombre de archivo ya existente).

El segundo método de utilería es **delete( ),** que borra el archivo de disco representado por el objeto **File** que invoca. El método tiene la forma:

boolean delete( )

También se puede utilizar **delete( )** para borrar un directorio si el directorio está vacío. **delete( )** devuelve **verdadero** si borra el archivo y **falso** si no ha sido posible borrarlo.

A continuación se listan algunos otros métodos de la clase **File** que encontrará de gran utilidad.

Método	Descripción
void deleteOnExit( )	Borra el archivo asociado con el objeto que invoca cuando la Máquina Virtual de Java termina.
long getFreeSpace( )	Regresa el número de bytes libres para almacenamiento disponibles en la partición asociada con el objeto que invoca. (Agregada por Java SE 6).
long getTotalSpace( )	Regresa la capacidad de almacenamiento de la partición asociada con el objeto que invoca. (Agregada por Java SE 6).

Método	Descripción
long getUtilizableSpace( )	Regresa el número de bytes libres que pueden ser utilizados para almacenamiento en la partición asociada con el objeto (Agregado por Java SE 6).
boolean isHidden( )	Devuelve verdadero si el archivo que invoca es un archivo oculto, y falso en caso contrario.
boolean setLastModified (long *m*)	Establece la fecha y hora del archivo que invoca al indicado por *m*, el cual representa el número de milisegundos transcurridos desde el 1 de enero de 1970 UTC (Tiempo Universal Coordinado).
boolean setReadOnly( )	Convierte el archivo que invoca en un archivo de sólo lectura.

También existen métodos para marcar archivos como de lectura, escritura y ejecutables. Dado que **File** Implementa la interfaz **Comparable**, el método **compareTo( )** también está soportado.

Directorios

Un directorio es un objeto **File** que contiene una lista de otros archivos y directorios. Cuando se crea un objeto **File** y es un directorio, el método **isDirectory( )** devolverá **verdadero**. En este caso se puede llamar al método **list( )** sobre ese objeto para extraer la lista de los archivos y directorios contenidos en él. Este método tiene dos formas. La primera es ésta:

```
String[ ] list( )
```

La lista de archivos se devuelve en un arreglo de objetos **String**.

El siguiente programa ilustra el uso de **list( )** para examinar el contenido de un directorio:

```
// Uso de directorios.
import java.io.File;

class DirLista {
  public static void main{String args[]) {
    String nomdir = "/java";
    File fl = new File (nomdir);

    if (fl.isDirectory()) {
      System.out.println ("Directorio de " + nomdir);
      String s[] = fl.list();

      for (int i = 0; i<s.length; i++) {
        File f = new File (nomdir + "/" + s[i]);
        if (f.isDirectory()) {
          System.out.println (s[i] + " es un directorio");
        } else {
          System.out.println(s[i] + " es un archivo");
        }
      }
    } else {
      System.out.println (nomdir + " no es un directorio");
    }
  }
}
```

Aquí tenemos una posible salida del programa anterior. Por supuesto, la salida será diferente dependiendo del contenido de cada directorio.

```
Directorio de /java
bin es un directorio
lib es un directorio
demo es un directorio
COPYRIGHT es un archivo
README es un archivo
index.html es un archivo
include es un directorio
src.zip es un archivo
src es un directorio
```

Uso de FilenameFilter

A menudo se desea limitar el número de archivos devueltos por el método **list( )** para incluir sólo los archivos cuyos nombres se ajusten a cierto patrón o *filtro*. Para hacer eso hay que utilizar una segunda forma de **list( )**, mostrada a continuación:

String[ ] list (FilenameFilter *objFF*)

En esta forma, *objFF* es un objeto de una clase que implementa la interfaz **FilenameFilter**.

FilenameFilter define un único método, **accept( )**, que se llama una vez por cada archivo en una lista. Su forma general es como sigue:

boolean accept (File *directorio*, String *nomArch*)

El método **accept( )** devuelve **verdadero** para los archivos en el *directorio* especificado por directorio y que deberían incluirse en la lista (esto es, aquellos que se ajustan al argumento *nomArch*), y devuelve **falso** para los archivos que deberían excluirse.

La clase **OnlyExt**, mostrada a continuación, implementa **FilenameFilter**. Se utilizará para modificar el programa anterior a modo de restringir la visibilidad de los archivos devueltos por **list( )** a aquellos con nombres que terminen con la extensión especificada como parámetro en la construcción del objeto.

```
import java.io.*;

public class OnlyExt implements FilenameFilter {
  String ext;

  public OnlyExt(String ext) {
    this.ext = "."+ ext;
  }

  public boolean accept(File dir, String nombre) {
    return name.endsWith(ext);
  }
}
```

A continuación se muestra el programa modificado de listado de un directorio. Ahora sólo mostrará archivos con la extensión .html.

```
// Directorio de archivos .HTML
import java.io.*;
```

```
class DirListOnly {
  public static void main (String args[]) {
    String nomdir = "/java";
    File fl = new File (nomdir);
    FilenameFilter only = new OnlyExt("html");
    String s[] = fl.list(only);

    for (int i=0; i < s.length; i++) {
      System.out.println(s[i]);
    }
  }
}
```


La alternativa listFiles( )

Existe una variación del método **list( )** llamada **listFiles( )**, la cual podría resultar útil. Las formas de **listFiles( )** son:

File[ ] listFiles( )
File[ ] listFiles(FilenameFilter *objFF*)
File[ ] listFiles(FileFilter *objF*)

Estos métodos devuelven la lista de archivos como un arreglo de objetos **File** en lugar de cadenas de texto. El primer método devuelve todos los archivos, y el segundo devuelve sólo los que satisfacen el filtro **FilenameFilter** especificado. Salvo por devolver un arreglo de objetos **File**, estas dos versiones de **listFiles( )** funcionan como sus métodos **list( )** equivalentes.

La tercera versión de **listFiles( )** devuelve aquellos archivos con rutas que satisfacen el filtro **FileFilter** especificado. **FileFilter** define un único método, **accept( )**, que se llama una vez para cada archivo de una lista. Su forma general es la siguiente:

boolean accept(File *rutaArch*)

El método **accept( )** devuelve **verdadero** para archivos que deberían incluirse en la lista (esto es, los que se ajustan al argumento *rutaArch*),y **falso** para los que deberían excluirse.

Creación de directorios

Otros dos métodos útiles de la clase File son los métodos **mkdir( )** y **mkdirs( )**. El método **mkdir( )** crea un directorio, devolviendo **verdadero** si hay éxito y **falso** si falla. Si falla indica que el directorio en la ruta especificada en el objeto **File** ya existe, o que el directorio no se puede crear porque aún no existe toda la ruta de directorios. Para crear un directorio para el que no existe una ruta, debe utilizarse el método **mkdirs( )**, el cual crea tanto el directorio como todos sus directorios padre.

Las interfaces Closeable y Flushable

Recientemente (con la versión de JDK 5), dos interfaces fueron agregadas a **java.io** dos nuevas interfaces: **Closeable** y **Flushable**. Estas interfaces son implementadas por muchas de las clases de E/S. Su inclusión no agrega nuevas funcionalidades a las clases de flujos. Simplemente ofrecen una forma uniforme de especificar que un flujo puede ser cerrado o limpiado.

Los objetos de una clase que implementa la interfaz **Closeable** pueden ser cerrados. La clase define el método **close( )** mostrado a continuación:

```
void close( ) throws IOException
```

Este método cierra el flujo que invoca, liberando cualquier recurso que pudiera ser retenido. Esta interfaz es implementada por todas las clases de E/S que abren un flujo que pueda ser cerrado.

Los objetos de una clase que implementan la interfaz **Flushable** puede forzar la escritura del contenido de un buffer en el flujo al cual el objeto está adherido. Las clases definen al método **flush( )** como se muestra a continuación:

```
void flush( ) throws IOException
```

Limpiar un flujo causa que la información contenida en un búfer sea físicamente escrita en el dispositivo subyacente. Esta interfaz está implementada por todas las clases de E/S que escriben datos en flujos.

Las clases Stream

La E/S de Java basada en flujos se construye sobre cuatro clases abstractas: **InputStream**, **OutputStream**, **Reader** y **Writer**. Estas clases se trataron brevemente en el Capítulo 13. Se utilizan para crear varias subclases concretas que trabajan con flujos. Aunque los programas lleven a cabo sus operaciones de E/S a través de subclases concretas, las clases del nivel superior definen la funcionalidad básica común a todas las clases que trabajan con flujos.

InputStream y **OutputStream** están diseñadas para flujos de bytes. **Reader** y **Writer** están diseñadas para flujos de caracteres. Las clases de flujos de bytes y las clases de flujos de caracteres forman jerarquías separadas. En general se deben utilizar las clases de flujos de caracteres al trabajar con cadenas o caracteres, y las de flujos de bytes al trabajar con bytes u otros objetos binarios.

En lo que queda de capítulo examinaremos tanto los flujos orientados a bytes como los orientados a caracteres.

Los flujos de Bytes

Las clases de flujos de bytes proporcionan un rico entorno para gestionar E/S orientadas a bytes. Un flujo de bytes se puede utilizar con cualquier tipo de objeto, incluyendo datos binarios. Esta versatilidad hace a los flujos de bytes importantes para muchos tipos de programas. Dado que las clases de flujos de bytes son subclases de **InputStream** y **OutputStream**, comenzaremos analizando estas clases.

InputStream

InputStream es una clase abstracta que define el modelo de Java para entradas por flujos de bytes. Esta clase implementa de la interfaz **Closeable**. La mayoría de los métodos de esta clase producen una excepción de tipo IOException cuando ocurre algún error. (Las excepciones son **mark( )** y **markSupported( )**). La Tabla 19-1 muestra los métodos de **InputStream**.

OutputStream

OutputStream es una clase abstracta que define los flujos de bytes de salida. Esta clase implementa las interfaces **Closeable** y **Flushable**. La mayoría de los métodos de esta clase devuelven un valor **void** y producen una excepción de tipo **IOException** en caso de errores. (Las

excepciones son **mark( )** y **markSupported( )**). La Tabla 19-2 muestra los métodos en la clase **OutputStream**.

Método	Descripción
int available( )	Devuelve el número de bytes de entrada actualmente disponibles para lectura.
void close( )	Cierra la fuente de entrada. Los intentos de lectura posteriores producirán una **IOException**.
void mark(int *numBytes*)	Coloca en el punto actual del flujo de entrada una marca que se mantendrá válida hasta que se lean *numBytes*.
boolean markSupported( )	Devuelve **verdadero** si **mark( )/reset( )** son soportadas por el flujo que invoca.
int read( )	Devuelve una representación como número entero del siguiente byte de entrada disponible. Se devuelve –1 cuando se ha llegado al final del archivo.
int read(byte *bufer*[ ])	Intenta leer hasta *bufer.length* bytes de *bufer* y devuelve el número actual de bytes que fueron leídos con éxito. Se devuelve –1 cuando se ha llegado al final del archivo.
int read(byte *bufer*[ ], int *desplazo*, int *numBytes*)	Intenta leer hasta *numBytes* bytes de *bufer* comenzando en *bufer[desplazo]*. Devuelve el número de bytes leídos con éxito. Se devuelve –1 cuando se ha alcanzado el final del archivo.
void reset( )	Inicializa el apuntador de la entrada a la marca previamente establecida.
long skip(long *numBytes*)	Ignora (esto es, se salta) *numBytes* bytes de la entrada y devuelve el número de bytes ignorados.

TABLA 19-1 Los métodos definidos por **InputStream**

NOTA *La mayoría de los métodos descritos en las Tablas 19-1 y 19-2 son implementados por las subclases de* ***InputStream*** *y* ***OutputStream****. Los métodos* ***mark( )*** *y* ***reset( )*** *son excepciones; observe su uso o su ausencia en cada subclase de las siguientes discusiones*

Método	Descripción
void close( )	Cierra el flujo de salida. Intentos posteriores de escritura producirán una **IOException**.
void flush( )	Finaliza el estado de la salida para que los búferes se limpien. Esto es, vacía los búferes de salida enviando su contenido al flujo.
void write(int *b*)	Escribe un único byte en un flujo de salida. Nótese que el parámetro es un **int**, lo que permite llamar a **write( )** con expresiones sin tener que volver a convertirlas a **byte**.
void write(byte *bufer*[ ])	Escribe una arreglo de bytes completo en un flujo de salida.
void write(byte *bufer*[ ], int *desplazo*, int *numBytes*)	Escribe un subintervalo de *numBytes* bytes en el arreglo llamado *bufer*, comenzando en *bufer[desplazado]*.

TABLA 19-2 Los métodos definidos por **OutputStream**

FileInputStream

La clase **FileInputStream** crea un flujo de entrada **InputStream** que se puede utilizar para leer bytes de un archivo. Sus dos constructores más usuales son los siguientes:

FileInputStream(String *rutaArch*)
FileInputStream(File *objArch*)

Cualquiera de los dos puede producir una excepción **FileNotFoundException**. Aquí, *rutaArch* es la ruta completa del archivo, y *objFile* es un objeto **File** que describe al archivo.

El siguiente ejemplo crea dos flujos **FileInputStream** que utilizan el mismo archivo en disco y cada uno de los dos constructores:

```
FileInputStream f0 = new FileInputStream ("/autoexec.bat")
File f = new File ("/autoexec.bat");
FileInputStream fl = new FileInputStream(f) ;
```

Aunque el primer constructor es probablemente el más utilizado, el segundo permite examinar de cerca el archivo utilizando los métodos de la clase **File** antes de asociarlo a un flujo de entrada. Cuando se crea un **FileInputStream**, también se abre para lectura. **FileInputStream** sobrescribe seis de los métodos de la clase abstracta **InputStream**. Los métodos **mark( )** y **reset( )** no se sobrescriben, y cualquier intento de utilizar **reset( )** sobre un **FileInputStream** generará una **IOException**.

El siguiente ejemplo muestra cómo leer un único byte, un arreglo de bytes y un subrango de un arreglo de bytes. También ilustra cómo utilizar el método **available( )** para determinar el número de bytes restantes, y cómo utilizar el método **skip( )** para saltarse los bytes no deseados. El programa lee su propio archivo fuente, que debe estar en el directorio actual.

```
// Ejemplo de FileInputStream.
import java.io.*;

class FileInputStreamDemo {
  public static void main (String args[]) throws Exception {
    int size;
    InputStream f =
      new FileInputStream ("FileInputStreamDemo.java");

    System.out.println ("Total de bytes disponibles: " +
                        (size = f.available()));
    int n = size/40;
    System.out.println ("Los primeros" + n +
                        " bytes del archivo se leen en una operación read()");
    for (int i=0; i < n; i++) {
      System.out.print ((char) f.read());
    }
    System.out.println("\nTotal disponible todavía: " + f.available());
    System.out.println("Leyendo los siguientes" + n +
                       " bytes con una llamada a read (b[])");
    byte b [] = new byte [n];
    if (f.read (b) != n) {
      System.err.println ( "No se pueden leer " + n + " bytes.");
    }
    System.out.println(new String(b, 0, n));
```

```
    System.out.println("\nTotal disponible todavía: " + (size = f.available()));
    System.out.println("Omite la mitad de los bytes restantes con skip()");
    f.skip (size/2) ;
    System.out.println ("Total disponible todavía: " + f.available ());
    System.out.println("Leyendo " + n/2 + " bytes del final del arreglo");
    if (f.read(b, n/2, n/2) != n/2) {
      System.err.println ("No se pueden leer " + n/2 + " bytes.");
    }
    System.out.println(new String(b, 0, b.length));
    System.out.println("\nTotal disponible aún: " + f.available());
    f.close () ;
  }
}
```

La salida producida por este programa es la siguiente:

```
Total de bytes disponibles: 1319
Los primeros 32 bytes del archivo se leen en una operación read( )
import java.io.*;

class FileInp

Total disponible todavía: 1287
Leyendo los siguientes32 bytes con una llamada a read (b[])
utStreamDemo {
public static vo
Total disponible todavía: 1255
Omite la mitad de los bytes restantes con skip( )
Total disponible todavía: 628
Leyendo 16 bytes del final del arreglo
utStreamDemo {
se pueden leer "

Total disponible aún: 612
```

Este ingenioso ejemplo ilustra cómo leer de tres formas, saltarse parte de la entrada, e inspeccionar la cantidad de datos disponible en un flujo.

NOTA *El ejemplo anterior (y en los otros ejemplos en este capítulo) manejan cualquier excepción de E/S que pudieran ocurrir lanzando la excepción* ***IOException*** *fuera de* ***main( )****, lo que significa que las excepciones son gestionadas por la JVM. Esto está bien para programas simples de demostración (y para pequeños programas de utilería que se escriban para uso propio), pero para aplicaciones comerciales normalmente se necesitará manejar las excepciones de E/S dentro del propio programa.*

FileOutputStream

FileOutputStream crea un **OutputStream** que se puede utilizar para escribir bytes en un archivo. Sus constructores más utilizados son estos:

FileOutputStream(String *rutaArchivo*)
FileOutputStream(File *objArchivo*)

FileOutputStream(String *rutaArchivo*, boolean *append*)
FileOutputStream(File *objArchivo*, boolean *append*)

Los tres pueden lanzar una **FileNotFoundException**. Aquí, *rutaArchivo* es la ruta completa del archivo, y *objArchivo* es un objeto **File** que describe al archivo. Si *append* es **verdadero**, el archivo se abre en modo de agregar.

La creación de un flujo tipo **FileOutputStream** no depende de que el archivo ya exista. Al crear el objeto, **FileOutputStream** creará el archivo antes de abrirlo como salida. En caso de intentar abrir un archivo de sólo lectura, se produce una **IOException**.

El siguiente ejemplo crea un búfer de bytes haciendo primero una cadena **String** y luego utilizando el método **getBytes( )** para obtener el arreglo de bytes equivalente. Entonces crea tres archivos. El primero, **filel.txt**, contendrá una letra si y otra no de la muestra. El segundo, **file2.txt**, contendrá todos los bytes. El tercero y último, **file3.txt**, contendrá sólo el último cuarto.

```
// Ejemplo de FileOutputStream.
import java.io.*;

class FileOutputStreamDemo {
  public static void main(String args[]) throws Exception {
    String source = "Ahora es el momento de que los hombres buenos\n"
      + " vengan a ayudar a su país\n"
      + " y paguen sus impuestos.";
    byte buf[] = source.getBytes();
    OutputStream f0 = new FileOutputStream("filel.txt");
    for (int i=0; i < buf.length; i += 2) {
      f0.write(buf[i]);
    }
    f0 .close () ;

    OutputStream fl = new FileOutputStream("file2.txt");
    f1.write(buf);
    f1. close ();

    OutputStream f2 = new FileOutputStream("file3.txt");
    f2.write(buf, buf.length-buf.length/4,buf.length/4);
    f2.close();
  }
}
```

A continuación se muestran los contenidos de cada archivo tras correr este programa. Primero, **file1.txt:**

```
Aoae lrmnnod u o obe uns egnaaua upí
ypge u muso.
```

Ahora, **file2.txt:**

```
Ahora es el momento de que los hombres buenos
  vengan a ayudar a su país
  y paguen sus impuestos.
```

Finalmente, **file3.txt**:

```
y paguen sus impuestos.
```

ByteArrayInputStream

ByteArrayInputStream es una implementación de un flujo de entrada que utiliza un arreglo de bytes como fuente. Esta clase tiene dos constructores, cada uno de los cuales requiere un arreglo de bytes que proporcione la fuente de datos:

ByteArrayInputStream(byte *arreglo*[ ])
ByteArrayInputStream(byte *arreglo*[ ], int *inicio*, int *numBytes*)

Aquí, *arreglo* es la fuente de entrada. El segundo constructor crea un **InputStream** a partir de un subconjunto del arreglo de bytes que *comienza* con el carácter ubicado en el índice indicado en inicio y tiene *numBytes* bytes de largo.

El siguiente ejemplo crea un par de flujos **ByteArrayInputStream**, inicializándolos con la representación en bytes del abecedario:

```
// Ejemplo con ByteArrayInputStream.
import java.io.*;

class ByteArrayInputStreamDemo {
  public static void main (String args[]) throws IOException {
    String tmp = "abcdefghijklmnñopqrstuvwxyz" ;
    byte b[] = tmp.getBytes( );
    ByteArrayInputStream entrada1 = new ByteArrayInputStream(b);
    ByteArrayInputStream entrada2 = new ByteArrayInputStream(b, 0,3);
  }
}
```

El objeto **entrada1** contiene el abecedario completo en minúsculas, mientras que **entrada2** contiene sólo las tres primeras letras.

Un **ByteArrayInputStream** implementa tanto **mark( )** como **reset( )**. Sin embargo, si **mark( )** no ha sido llamado, entonces **reset( )** coloca el apuntador del flujo al principio de él, que en este caso es el comienzo del arreglo de bytes pasado al constructor. El siguiente ejemplo muestra cómo utilizar el método **reset( )** para leer dos veces la misma entrada. En este caso, se leen e imprimen las letras "abc" una vez en minúsculas y luego de nuevo en mayúsculas.

```
import java.io.*;

class ByteArrayInputStreamReset {
  public static void main(String args[]) throws IOExeeption {
    String tmp = "abc";
    byte b[] = tmp.getBytes ( );
    ByteArrayInputStream in = new ByteArrayInputStream(b);

    for (int i=0; i<2; i++) {
      int c;
      while ((c = in.read( )) != -1) {
        if (i == 0) {
          System.out.print((char) c);
        } else {
          System.out.print(Character.toUpperCase((char) c));
        }
      }
```

```
      System.out.println( );
      in.reset( );
    }
  }
}
```

Este ejemplo primero lee cada carácter del flujo y lo imprime tal cual, en minúsculas. Luego reinicia el flujo y comienza de nuevo a leer, esta vez convirtiendo cada carácter a mayúscula antes de imprimir. Aquí está la salida:

```
abc
ABC
```

ByteArrayOutputStream

ByteArrayOutputStream es una implementación de un flujo de salida que utiliza un arreglo de bytes como destino. **ByteArrayOutputStream** tiene dos constructores, aquí mostrados:

```
ByteArrayOutputStream()
ByteArrayOutputStream(int numBytes)
```

En la primera forma se crea un búfer de 32 bytes. En la segunda se crea un búfer de tamaño especificado en *numBytes*. El búfer se guarda en el campo protegido **buf** de **ByteArrayOutputStream**. El tamaño del búfer aumenta automáticamente cuando es necesario. El número de bytes en el búfer se guarda en campo protegido **count** de **ByteArrayOutputStream**.

El siguiente ejemplo ilustra la clase **ByteArrayOutputStream**:

```
// Ejemplo de ByteArrayOutputStream.
import java.io.*;

class ByteArrayOutputStreamDemo {
  public static void main(String args[]) throws IOExeeption {
    ByteArrayOutputStream f = new ByteArrayOutputStream();
    String s = "Esto debería terminar en el arreglo";
    byte buf [] = s.getBytes();
    f.write(buf) ;
    System.out.println("Búfer como cadena");
    System.out.println(f.toString());
    System.out.println("Al arreglo");
    byte b [] = f.toByteArray();
    for (int i=0; i<b.length; i++) {
      System.out.print ((char) b[i]);
    }
    System.out.println("\n A un OutputStream()");
    OutputStream f2 = new FileOutputStream("test.txt");

    f.writeTo (f2);
    f2.close ();
    System.out.println("Reinicializando");
    f.reset () ;
    for (int i=0, i<3, i++)
      f.write('X');
    System.out.println(f.toString( ));
  }
}
```

Al correr el programa se crea la siguiente salida. Nótese cómo tras cada llamada a **reset( )**, las tres *X* finales están al principio.

```
Búfer como cadena
Esto debería terminar en el arreglo
Al arreglo
Esto debería terminar en el arreglo
A un OutputStream( )
Reinicializando
XXX
```

Este ejemplo aprovecha la comodidad del método **writeTo( )** para escribir los contenidos de **f** en el archivo test.txt. Revisando el contenido del archivo **text.txt** creado en el ejemplo anterior muestra el resultado esperado:

```
Esto debería terminar en el arreglo
```


Flujos de bytes filtrados

Los *flujos filtrados* son simplemente envolturas alrededor de flujos de entrada o salida que proporcionan de forma transparente cierto grado extendido de funcionalidad. A estos flujos acceden típicamente métodos que esperan un flujo genérico, el cual es una superclase de los flujos filtrados. Extensiones típicas son los búferes, las traducciones de caracteres y las traducciones de datos brutos. Los flujos de bytes filtrados son **FilterInputStream** y **FilterOutputStream**. Sus constructores son:

FilterOutputStream(OutputStream *os*)
FilterInputStream(InputStream *is*)

Los métodos proporcionados en estas clases son idénticos a los de **InputStream** y **OutputStream**.

Flujos de bytes con búfer

Para los flujos orientados a bytes, un *flujo con búfer* extiende una clase de flujo filtrado y le añade un búfer de memoria. Este búfer permite a Java hacer operaciones de E/S sobre más de un byte a la vez, mejorando así el rendimiento. Al estar el búfer disponible, se hace posible saltar, marcar o reinicializar el flujo. Las clases de flujos de bytes con búfer son **BufferedInputStream** y **BufferedOutputStream**. La clase **PushbackInputStream** también implementa un flujo con búfer

BufferedInputStream

Dotar de un búfer a la E/S es una forma común de optimizar el rendimiento. La clase de Java **BufferedInputStream** permite "envolver" cualquier **InputStream** en un flujo con búfer y así conseguir esta mejora del rendimiento.

BufferedInputStream tiene dos constructores:

BufferedInputStream(InputStream *flujoEntrada*)
BufferedInputStream(InputStream *flujoEntrada*, int *tamañoBufer*)

La primera forma crea un flujo con búfer utilizando un tamaño de búfer por omisión. En la segunda, el tamaño del búfer se pasa en *tamañoBufer*. Utilizar tamaños que sean múltiplos de la

página de memoria o del bloque de disco puede mejorar significativamente el rendimiento. Esto, sin embargo, es dependiente de la implementación. El tamaño óptimo de búfer generalmente depende del sistema operativo de la computadora, la cantidad de memoria disponible, y de cómo está configurada la máquina. Para hacer un buen uso de los búferes no hace falta necesariamente todo ese grado de sofisticación. Una buena estimación del tamaño es alrededor de 8,192 bytes; y añadir un búfer, aunque sea pequeño, a un flujo de E/S siempre es una buena idea. De esa forma, el sistema de bajo nivel puede leer bloques de datos del disco o de la red y almacenar los resultados en el búfer. Incluso si se está leyendo los bytes de uno en uno desde el **InputStream**, se estará manipulando memoria rápida la mayor parte del tiempo.

Dotar de un búfer a un flujo de entrada también proporciona la base necesaria para permitir el movimiento hacia atrás en el flujo del búfer disponible. Además de los métodos **read( )** y **skip( )** son implementados en cualquier **InputStream**, **BufferedInputStream** también soporta los métodos **mark( )** y **reset( )**. Esto se refleja en el hecho de que **BufferedInputStream**.**mark Supported( )** devuelve verdadero.

El siguiente ejemplo presenta una situación en que se puede utilizar **mark( )** para recordar dónde estamos en un flujo de entrada, y luego utilizar **reset( )** para volver a ese punto. Este ejemplo analiza un flujo buscando la referencia asociada al símbolo de copyright en HTML. Tal referencia comienza con un signo "&" y termina con un punto y coma ";" sin que haya ningún espacio en blanco de por medio. La entrada del ejemplo tiene dos signos "&" para mostrar dónde se produce el **reset( )** y dónde no.

```
// Uso de una entrada con búfer.
import java.io.*;

class BufferedInputStreamDemo {
  public static void main(String args[]) throws IOExeeption {
    String s = "Esto &copy; es un símbolo de copyright " +
      "pero esto &copy no lo es.\n";
    byte buf[] = s.getBytes( );
    ByteArrayInputStream in = new ByteArrayInputStream(buf);
    BufferedInputStream f = new BufferedInputStream(in);
    int c;
    boolean marcado = false;

    while ((c = f.read( )) != -1) {
      switch (c) {
      case '&':
        if (!marcado) {
          f.mark(32);
          marcado = true;
        } else {
          marcado = false;
        }
        break;
      case ';':
        if (marcado) {
          marcado = false;
          System.out.print("(c)");
        } else
          System.out.print((char) c);
        break;
      case' ':
```

```
        if (marcado) {
          marcado = false;
          f .reset ();
          System.out.print("&");
        } else
          System.out.print((char) c);
        break;
    default:
      if (!marcado)
          System.out.print( (char) c);
      break;
      }
    }
  }
}
```

Nótese que este ejemplo utiliza **mark(32)**, que conserva la marca para los siguientes 32 bytes leídos (lo cual es suficiente para todas las referencias a entidades). Aquí está la salida producida por este programa:

```
Esto (c) es un símbolo de copyright pero esto &copy no lo es.
```

BufferedOutputStream

La clase **BufferedOutputStream** es similar a cualquier **OutputStream**, salvo por un método añadido **flush( )** el cual se utiliza para asegurar que los búferes de datos sean escritos realmente al dispositivo de salida físico. Dado que la idea de un **BufferedOutputStream** es mejorar el rendimiento reduciendo el número de veces que el sistema escribe realmente datos, lo normal es que sea necesario llamar a **flush( )** para hacer que los datos del búfer se escriban inmediatamente.

Al contrario de lo que ocurre con la entrada con búfer, dotar de un búfer a la salida no proporciona funcionalidad adicional. Los búferes para salida en Java existen sólo para mejorar el rendimiento. Aquí están los dos constructores disponibles:

BufferedOutputStream(OutputStream *flujoSalida*)
BufferedOutputStream(OutputStream *flujoSalida*, int *tamañoBufer*)

La primera forma crea un flujo con búfer utilizando el tamaño del búfer por omisión. En la segunda forma, el tamaño del búfer se pasa en *tamañoBufer*.

PushbackInputStream

Un uso nuevo de los búferes es la implementación de la devolución (*pushback*). La *devolución* se utiliza sobre un flujo de entrada para permitir que un byte se lea y luego se devuelva al flujo. La clase **PushbackInputStream** implementa esta idea. Proporcionando un mecanismo para "inspeccionar" sin alterar lo que llega por un flujo de entrada.

PushbackInputStream tiene los siguientes constructores:

PushbackInputStream(InputStream *flujoEntrada*)
PushbackInputStream(InputStream *flujoEntrada*, int *numBytes*)

La primera forma crea un objeto de flujo que permite devolver un byte al flujo de entrada. La segunda forma crea un flujo con un búfer de devolución de longitud *numBytes*. Esto permite devolver múltiples bytes al flujo de entrada.

Además de los métodos de **InputStream** habituales, **PushbackInputStream** incluye **unread( )**, aquí mostrado:

void unread(int *cars*)
void unread(byte *bufer[])*
void unread(byte *bufer*, int *desplaza*, int *numCars*)

La primera forma devuelve el byte menos significativo de *cars*. Este será el siguiente byte devuelto por una subsiguiente llamada a **read( )**. La segunda forma devuelve los bytes en el *búfer*. La tercera forma devuelve *numCars* bytes del *búfer* comenzando en *desplaza*. Si se intenta devolver un byte estando lleno el búfer de devolución, se produce una excepción **IOException**.

Aquí hay un ejemplo que muestra cómo un analizador de un lenguaje de programación podría utilizar un **PushbackInputStream** y **unread( )** para diferenciar el operador **= =** de comparación del operador **=** de asignación:

```
// Ejemplo de unread().
import java.io.*;

class PushbackInputStreamDemo {
  public static void main(String args[]) throws IOExeeption {
    String s = "if (a = = 4) a = 0 ; \n";
    byte buf[] = s.getBytes();
    ByteArrayInputStream in = new ByteArrayInputStream(buf);
    PushbackInputStream f = new PushbackInputStream(in);
    int c;

    while ((c = f.read()) != -1) {
      switch(c) {
      case '=':
        if ((c = f.read()) == '=')
          System.out.print(".eq.");
        else {
          System.out.print("<-");
          f.unread(c) ;
        }
        break;
    default:
      System.out.print((char) c);
      break;
      }
    }
  }
}
```

Aquí está la salida de este ejemplo. Nótese que **= =** ha sido remplazado por "**.eq.**"y **=** por "**<-**".

```
  if (a .eq. 4) a <- 0;
```

PRECAUCIÓN ***PushbackInputStream*** *tiene el efecto colateral de invalidar los métodos* ***mark( )*** *o* ***reset( )*** *del* ***InputStream*** *utilizado para crearlo. Utilice* ***markSupported( )*** *para comprobar cualquier flujo sobre el que se vayan a utilizar los métodos* ***mark( ) y reset( )****.*

SequenceInputStream

La clase **SequenceInputStream** permite concatenar múltiples flujos del tipo **InputStream**. La construcción de un **SequenceInputStream** es diferente de la de cualquier otro **InputStream**. Un constructor **SequenceInputStream** utiliza como argumento un par de flujos **InputStream** o una **Enumeration** de **InputStream** como argumentos:

SequenceInputStream(InputStream *primero*, InputStream *segundo*)
SequenceInputStream(Enumeration <? extends InputStream> *StreamEnum*)

Operacionalmente, la clase completa las peticiones de lectura del primer **InputStream** hasta que se acaba, y después cambia al segundo. En el caso de una **Enumeration**, continuará a través de todos los flujos **InputStream** hasta que se llegue al final del último.

Este sencillo ejemplo utiliza un **SequenceInputStream** para sacar los contenidos de dos archivos:

```
// Ejemplo de entrada secuencial.
import java.io.*;
import java.util.*;

class InputStreamEnumerator implements Enumeration <FileInputStream>{
  private Enumeration <String> files;

  public InputStreamEnumerator(Vector <String> files) {
    this.files = files.elements();
  }

  public boolean hasMoreElements() {
    return files.hasMoreElements();
  }

  public Object nextElement() {
    try{
      return new FileInputStream(files.nextElement().toString());
    } catch (IOException e) {
      return null;
    }
  }
}

class SequenceInputStreamDemo {
  public static void main(String args[])
    throws IOException {

    int c;
    Vector <String> files = new Vector<String>();

    files.addElement("/autoexec.bat");
    files.addElement("/config.sys");
    InputStreamEnumerator e = new InputStreamEnumerator(files);
    InputStream input = new SequenceInputStream(e);

    while ((c = input. read () ) != -1) {
      System.out.print((char) c);
    }
```

```
    input.close ( ) ;
  }
}
```

Este ejemplo crea un **Vector** y luego le añade dos nombres de archivos. Le pasa ese vector de nombres a la clase **InputStreamEnumerator**, que está diseñada para proporcionar un envoltorio al vector donde los elementos devueltos no son nombres de archivos, sino flujos **FileInputStream** abiertos sobre esos nombres. El flujo **SequenceInputStream** abre cada archivo uno a uno, y este ejemplo imprime los contenidos de ambos archivos.

PrintStream

La clase **PrintStream** ofrece todas las capacidades de salida y formato que hemos estado utilizando con el descriptor de archivos de **System**, **System.out**, desde el principio del libro. Esto hace a **PrintStream** una de las clases de Java más frecuentemente utilizadas. Implementa las interfaces **Appendable**, **Closeable** y **Flushable**.

PrintStream define varios constructores. Los que se muestran a continuación han sido especificados desde el principio:

PrintStream(OutputStream *flujoSalida)*
PrintStream(OutputStream *flujoSalida*, boolean *vaciarSiNuevaLinea*)
PrintStream(OutputStream *flujoSalida*, boolean *vaciarSiNuevaLinea*,
String *ConjuntoCars*)

Donde, *flujoSalida* especifica abrir un **OutputStream** que recibirá una salida. El parámetro *vaciarSiNuevaLinea* controla si el flujo de salida es automáticamente limpiado cada vez que se envía un carácter de nueva línea (**\n**) o si un arreglo de byte es escrito, o cuando **println( )** es llamado. Si *vaciarSiNuevaLinea* es verdadero, el vaciado se lleva a cabo automáticamente. Si es falso, el vaciado no es automático. El primer constructor no vacía automáticamente. Se puede especificar un carácter de codificación pasando su nombre en *ConjutoCars*.

El siguiente conjunto de constructores, proporcionan una forma fácil de construir un **PrintStream** que escriba su salida a un archivo.

PrintStream(File s*alidaArchivo*) throws FileNotFoundException
PrintStream(File *salidaArchivo*, String *charSet*)
throws FileNotFoundException, UnsupportedEncodingException

PrintStream(File s*alidaNomArchivo*) throws FileNotFoundException
PrintStream(File *salidaNomArchivo*, String *charSet*)
throws FileNotFoundException, UnsupportedEncodingException

Estos constructores permiten a **PrintStream** ser creado ya sea por un objeto **File** o especificando el nombre de un archivo. En cualquier caso, el archivo es creado automáticamente. Si existe un archivo con el mismo nombre se destruye. Una vez creado, el objeto **PrintStream** dirige la salida al archivo especificado. Se puede especificar un carácter de codificación pasando su nombre *charSet*.

PrintStream soporta los métodos **print( )** y **println( )** para todos los tipos, incluyendo **Object**. Si un argumento no es de tipo primitivo, los métodos de **PrintStream** llamarán al método **toString( )** del objeto y luego desplegarán el resultado.

Recientemente (con la versión de JDK 5), el método **printf( )** fue añadido a **PrintStream**. Esto permite especificar el formato preciso de los datos que serán escritos. El método **printf( )** utiliza la clase **Formatter** (descrita en el Capítulo 18) para dar formato a los datos. El método luego escribe esta información al flujo llamante. Aunque se puede dar formato manualmente, utilizando la clase **Formatter** directamente, **printf( )** automatiza el proceso. Es también equivalente a la función **printf( )** de C/C++, lo cual facilita la conversión del código de C/C++ existente a código de Java. Francamente, **printf( )** es una adición bienvenida al API de Java, dado que simplifica enormemente la salida de datos con formato a la consola.

El método **printf( )** tiene la siguiente forma general:

PrintStream printf(String *formatoString*, Object … *args*)

PrintStream printf(Locale *loc*, String *formatoString*, Object … *args*)

La primera versión escribe *args* a la salida estándar en el formato especificado por *formatoString*, utilizando el locale por omisión. La segunda versión, te permite especificar un locale. Ambos regresan al **PrintStream** que invoca.

En general, **printf( )** funciona de una manera similar al método **format( )** especificado por **Formatter**. El parámetro *formatoString* consiste de dos tipos de datos. El primer tipo está compuesto de simples caracteres que son copiados al búfer de salida. El segundo tipo contiene especificadores de formato que definen la forma en que los subsecuentes argumentos, especificados por *args*, son mostrados. Para información detallada sobre dar formato a la salida, incluyendo una descripción de los especificadores de formatos, véase la descripción de la clase **Formatter** en el Capítulo 18.

Dado que **System.out** es un **PrintStream**, se puede llamar **printf( )** sobre **System.out**. Así, **printf( )** puede ser utilizado en lugar de **println( )** para escribir en la consola en cualquier momento que se desee tener una salida con formato. Por ejemplo, el siguiente programa utiliza **printf( )** para mostrar valores numéricos en varios formatos. En el pasado, tal formato requería un poco de trabajo. Con la adición de **printf( )**, esto se ha convertido en una tarea fácil.

```
// Ejemplo de printf( )

class PrintfDemo {
  public static void main(String args[]) {
    System.out.println ("Aquí se encuentran algunos valores numéricos " +
                        "en diferentes formatos.\n");

    System.out.printf("Varios formatos de enteros: ");
    System.out.printf("%d %(d %+d %05d\n", 3, -3, 3, 3);

    System.out.println();

    System.out.printf("Formato por omisión para punto flotante: %f\n",
                      1234567.123);
    System.out.printf("Punto Flotante con comas: %,f\n",
                      1234567.123);
    System.out.printf("Punto Flotante negativo: %,f\n",
                      -1234567.123);
    System.out.printf("Opción de punto flotante negativo: %, (f\n",
                      -1234567.123);

    System.out.println( );

    System.out.printf("Alineando valores positivos y negativos:\n");
    System.out.printf("% ,.2f\n% ,.2f\n",
```

```
                    1234567.123, -1234567.123);
  }
}
```

La salida se muestra a continuación:

```
Aquí se encuentran algunos valores numéricos

Varios formatos de enteros: 3 (3) +3 00003

Formato por omisión para punto flotante: 1234567.123000
Punto Flotante con comas: 1,234,567.123000
Punto Flotante negativo: -1,234,567.123000
Opción de punto flotante negativo: (1,234,567.123000)

Alineando valores positivos y negativos:
1,234,567.12
-1,234,567.12
```

PrintStream también define el método **format( )**, el cual tiene la siguiente forma general:

PrintStream format(String *fmtString,* Object … *args*)

PrintStream format(Locale *loc,* String *fmtString,* Object … *args*)

Funciona exactamente igual que **printf( ).**

DataOutputStream y DataInputStream

DataOutputStream y **DataInputStream** permite leer o escribir datos primitivos hacia o desde un flujo. Estas clases implementan las interfaces **DataOutput** y **DataInput** respectivamente. Estas interfaces definen métodos que convierten valores primitivos hacia o desde una secuencia de bytes. Estos flujos facilitan el almacenamiento de datos binarios en un archivo, tales como valores enteros o de punto flotante. Cada uno se examina a continuación.

DataOutputStream extiende de **FilterOutputSteam**, el cual extiende **OutputStream**. **DataOutputStream** define el siguiente constructor:

DataOutputStream (OutputStream *flujoSalida*)

Donde, *flujoSalida* especifica el flujo de salida sobre el cual los datos serán escritos

DataOutputStream soporta todos los métodos definidos por su superclases. Sin embargo, son los métodos definidos por la interfaz **DataOutput**, implementados por **DataOutputStream**, los que la hacen interesante. **DataOutput** define métodos que convierten valores de tipo primitivo en una secuencia de bytes y luego los escribe al flujo. A continuación un ejemplo de esos métodos.

final void writeDouble(double *valor*) throws IOException
final void writeBoolean(boolean *valor*) throws IOException
final void writeInt(int *valor*) throws IOException

Donde valor es el valor escrito en el flujo

DataInputStream es el complemento de **DataOutputStream**. Esta clase extiende de **FilterInputStream** que a su vez extiende de **InputStream**, e implementa la interfaz **DataInput**. A continuación el único constructor de la clase:

DataInputStream(InputStream *flujoEntrada*)

Donde, *flujoEntrada* especifica el flujo de entrada sobre el cual los datos serán leídos.

Como **DataOutputStream**, **DataInputStream** soporta todos los métodos de sus superclases, pero son los métodos de la interfaz **DataInput**, los que la hacen única. Esos métodos leen una secuencia de bytes y la convierten en valores de tipo primitivo. Aquí se presenta un ejemplo de esos métodos:

```
double readDouble( ) throws IOException
boolean readBoolean( ) throws IOException
int readInt( ) throws IOException
```

El siguiente programa demuestra el uso de **DataOutputStream** y **DataInputStream**:

```
import java.io.*;

class DataIODemo {
  public static void main(String args[])
    throws IOException {

    FileOutputStream fout = new FileOutputStream("Test.dat");
    DataOutputStream out = new DataOutputStream(fout);

    out.writeDouble(98.6);
    out.writeInt(1000);
    out.writeBoolean(true);

    out.close();

    FileInputStream fin = new FileInputStream("Prueba.dat");
    DataInputStream in = new DataInputStream(fin);

    double d = in.readDouble();
    int i = in.readInt();
    boolean b = in.readBoolean();

    System.out.println("A continuación se muestran los valores: "+
                       d + " " + i + " " + b) ;

    in.close ( ) ;
  }
}
```

La salida se muestra a continuación

```
A continuación se muestran los valores: 98.6 1000 true
```

RandomAccessFile

RandomAccessFile encapsula un archivo de acceso aleatorio. Esta clase no se derivada de **InputStream** u **OutputStream**, sino que implementa las interfaces **DataInput** y **DataOutput**, que definen los métodos básicos de E/S. **RandomAccessFile** implementa también la interfaz

Closeable. Además permite peticiones de posicionamiento, esto es, se puede posicionar el *apuntador del archivo* dentro del mismo archivo. Tiene estos dos constructores:

RandomAccessFile (File *objFile*, String *acceso*)
  throws FileNotFoundException

RandomAccessFile(String *nomArchivo*, String *acceso*)
  throws FileNotFoundException

En la primera forma, *objFile* indica el nombre del archivo que se ha de abrir como objeto **File**. En la segunda forma, el nombre del archivo se pasa en *nomArchivo*. En ambos casos, acceso determina qué tipo de *acceso* está permitido. Si es "r" el archivo se puede leer, pero no escribir. Si es "rw" entonces el archivo se abre en modo de lectura-escritura. Si es "rws" el archivo es abierto como lectura-escritura y todos los cambios realizados a los datos del archivo o metadatos serán escritos inmediatamente en el dispositivo físico. Si es "rwd", el archivo es abierto para operaciones de lectura y escritura y todos los cambios a los datos del archivo serán inmediatamente escritos al dispositivo físico.

El método **seek( )**, mostrado a continuación, se utiliza para establecer la posición actual del apuntador dentro del archivo:

void seek (long *nuevaPos*) throws IOException

Aquí, *nuevaPos* especifica la nueva posición, en bytes, del apuntador del archivo a partir del inicio del archivo. Tras una llamada a **seek( )**, la siguiente operación de lectura o escritura ocurrirá en la nueva posición del archivo.

RandomAccessFile implementa los métodos de entrada y salida estándar, que se pueden utilizar para leer y escribir archivos de acceso aleatorio. Y también incluye algunos métodos adicionales, uno es **setLength( )**. Ese método tiene la forma:

void setLength(long *t*) throws IOException

Este método establece la longitud del archivo que invoca a la especificada por *t*. Este método se puede utilizar para alargar o acortar un archivo. Si el archivo se alarga, el trozo añadido está indefinido.

Los flujos de caracteres

Aunque las clases de flujos de bytes proporcionan suficiente funcionalidad para gestionar cualquier tipo de operación de E/S, no pueden trabajar directamente con caracteres Unicode. Puesto que una de las principales finalidades de Java es dar soporte a la filosofía "escribir una vez, ejecutar en cualquier sitio", era necesario incluir soporte directo de E/S para caracteres. En esta sección se tratan varias de las clases de E/S de caracteres. Como ya se ha explicado, en la cumbre de las jerarquías de flujos de caracteres se encuentran las clases abstractas **Reader** y **Writer**. Comenzaremos con ellas.

NOTA *Como se ha visto en el Capítulo 13, las clases de E/S de caracteres fueron añadidas por la versión de Java 1.1. Debido a esto, es posible seguir encontrando código preexistente que utiliza flujos de bytes donde debería utilizar flujos de caracteres. Si se trabaja sobre este código, es una buena idea actualizarlo.*

Reader

Reader es una clase abstracta que define el modelo de Java para trabajar con flujos de entrada de caracteres. Ésta implementa las interfaces **Closeable** y **Readable**. Todos los métodos de esta clase (excepto **markSupported( )**) producirán una excepción de tipo **IOException** ante condiciones de error. La Tabla 19-3 es una sinopsis de los métodos de **Reader**.

Writer

Writer es una clase abstracta que define los flujos de salida de caracteres. Ésta implementa las interfaces **Closeable**, **Flushable** y **Appendable**. Todos los métodos de esta clase producen una excepción **IOException** en caso de errores. La Tabla 19-4 muestra una sinopsis de los métodos de **Writer**.

FileReader

La clase **FileReader** crea un **Reader** que se puede utilizar para leer los contenidos de un archivo. Sus dos constructores más comúnmente utilizados son:

FileReader(String *rutaArchivo*)
FileReader(File *objFile*)

Cualquiera de ellos puede producir una excepción de tipo **FileNotFoundException**. Aquí, *rutaArchivo* es la ruta completa de un archivo, y *objFile* es un objeto File que describe el archivo.

El siguiente ejemplo muestra cómo leer líneas de un archivo e imprimirlas al flujo de salida estándar. El programa lee su propio archivo fuente, que debe estar en el directorio actual.

```
// Ejemplo de FileReader.
import java.io.*;

class FileReaderDemo {
  public static void main(String args[]) throws Exception {
    FileReader fr = new FileReader ("FileReaderDemo.java");
    BufferedReader br = new BufferedReader(fr);
    String s;

    while((s = br.readLine()) != null) {
      System.out.println(s);
    }

    fr.close();
  }
}
```

FileWriter

FileWriter crea un **Writer** que se puede utilizar para escribir a un archivo. Sus constructores más comúnmente utilizados se muestran a continuación:

FileWriter(String *rutaArchivo*)
FileWriter(String *rutaArchivo*, boolean *añadir*)
FileWriter(File *objFile*)
FileWriter(File *objFile*, boolean *añadir*)

Estos métodos pueden producir una **IOException**. Aquí, *rutaArchivo* es la ruta del archivo, y *objFile* es un objeto **File** que describe al archivo. Si *añadir* es **true**, entonces la salida se añade al final del archivo.

Método	Descripción
abstract void close( )	Cierra la fuente de entrada. Intentos de lectura posteriores generarán una **IOException**.
void mark(int *numCars*)	Coloca en el punto actual del flujo de entrada una marca que permanecerá válida hasta que se hayan leído *numCars* caracteres.
boolean markSupported( )	Devuelve **true** si **mark( )/reset( )** están soportados en este flujo.
int read( )	Devuelve una representación como entero del siguiente carácter disponible en el flujo de entrada que invoca. Se devuelve –1 cuando se ha alcanzado el final del archivo.
int read(char *bufer*[ ])	Intenta leer hasta *bufer.length* caracteres de *bufer* y devuelve el número real de caracteres leídos con éxito. Se devuelve –1 cuando se ha alcanzado el final del archivo.
abstract int read(char *bufer*[ ], int *desplazo*, int *numCars*)	Intenta leer hasta *numCars* caracteres de *bufer* comenzando con *bufer*[*desplazo*], devuelve el número de caracteres leídos con éxito o –1 cuando se ha alcanzado el final del archivo.
boolean ready( )	Devuelve **verdadero** si la siguiente petición de entrada no tendrá que esperar. De lo contrario, devuelve **falso**.
void reset( )	Inicializa el apuntador de la entrada a la marca previamente establecida.
long skip(long *numCars*)	Se salta *numCars* caracteres de entrada, devolviendo el número de caracteres realmente saltados.

TABLA 19-3 Los métodos definidos por **Reader**

Método	Descripción
Writer append(char *ch*)	Anexa *ch* al final del flujo de salida que invoca. Regresa la referencia al flujo que invoca.
Writer append(CharSequence *chars*)	Anexa *chars* al final del flujo de salida que invoca. Regresa la referencia al flujo que invoca.
Writer append(CharSequence *chars*, int *begin*, int *end*)	Anexa el subrango de caracteres especificados por *begin* y *end*–1 al final del flujo de salida que invoca. Devuelve la referencia al flujo que invoca.
abstract void close( )	Cierra el flujo de entrada. Los intentos posteriores de escritura generarán una **IOException**.
abstract void flush( )	Finaliza el estado de la salida de modo que los búferes se limpien. Esto es, vacía los búferes de salida.
void write(int *car*)	Escribe un único carácter en el flujo de salida que invoca. Nótese que el parámetro es un **int**, lo que permite llamar a **write** con expresiones sin tener que convertirlas de nuevo a **char**.

TABLA 19-4 Los métodos definidos por **Writer**

Método	Descripción
void write(char *bufer*[ ])	Escribe un arreglo completo de caracteres en el flujo de salida que invoca.
abstract void write(char *bufer*[ ], int *desplazo*, int *numCars*)	Escribe un subintervalo de *numCars* caracteres del arreglo bufer, comenzando en *bufer*[*desplazo*], en el flujo de salida que invoca.
void write(String *cad*)	Escribe *cad* al flujo de salida que invoca.
void write(String *cad*, int *indice*, int *numCars*)	Escribe un subintervalo de *numCars* caracteres de la cadena *cad*, comenzando en el *índice* especificado.

TABLA 19-4 Los métodos definidos por **Writer** (*continuación*)

La creación de un **FileWriter** no depende de la existencia previa del archivo. **FileWriter** creará el archivo antes de abrirlo para salida cuando se crea el objeto. En caso de intentar abrir un archivo de sólo lectura, se produce una **IOException**.

El siguiente ejemplo es una versión con flujo de caracteres de un ejemplo mostrado antes, cuando se trató el **FileOutputStream**. Esta versión crea un búfer con caracteres de ejemplo creando primero una cadena y luego utilizando el método **getChars( )** para extraer el arreglo de caracteres equivalente. Luego crea tres archivos. El primero, **file1.txt**, contendrá uno de cada dos caracteres de la muestra. El segundo, **tile2.txt**, contendrá el conjunto completo de caracteres. Finalmente el tercero, **tile3.txt**, contendrá sólo el último cuarto de caracteres.

```
// Ejemplo de FileWriter.
import java.io.*;

class FileWriterDemo {
  public static void main String args[]) throws IOExeeption {
    String source = "Ahora es el momento de que los hombres buenos\n"
      + " vengan a ayudar a su país\n"
      + " y paguen sus impuestos.";
    char buffer[] = new char[source.length( )];
    source.getChars(0, source.length(), buffer, 0);

    FileWriter f0 = new FileWriter("filel.txt");
    for (int i=0; i < buffer.length; i += 2) {
      f0.write(buffer[i]);
    }
    f0.close();

    FileWriter f1 = new FileWriter ("file2. txt");
    fl.write(buffer);
    fl.close();

    FileWriter f2 = new FileWriter("file3.txt");

    f2.write(buffer,buffer.length-buffer.length/4,buffer.length/4);
    f2.close();
  }
}
```

CharArrayReader

CharArrayReader es una implementación de un flujo de entrada que utiliza como fuente un arreglo de caracteres. Esta clase tiene dos constructores, cada uno de los cuales requiere un arreglo de caracteres para proporcionar la fuente de datos:

CharArrayReader(char *arreglo*[ ])
CharArrayReader(char *arreglo*[ ], int *inicio*, int *numCars*)

Aquí, *arreglo* es la fuente de entrada. El segundo constructor crea un **Reader** a partir de un subconjunto del arreglo de caracteres que empieza con el carácter en la posición especificada por *inicio* y tiene *numCars* caracteres de longitud.

El siguiente ejemplo utiliza un par de **CharArrayReaders**:

```
// Ejemplo de CharArrayReader.
import java.io.*;

public class CharArrayReaderDemo {
  public static void main(String args[]) throws IOExeeption {
    String tmp = "abcdefghijklmnopqrstuvwxyz";
    int length = tmp.length();
    char c[] = new char[length];

    tmp.getChars(0, length, c, 0);
    CharArrayReader entrada1 = new CharArrayReader(c);
    CharArrayReader entrada2 = new CharArrayReader(c, 0, 5);

    int i;
    System.out.println("entrada1 es:");
    while((i = entrada1.read( )) != -1) {
      System.out.print((char)i);
    }
    System.out.println();

    System.out.println("entrada2 es:");
    while ((i = entrada2.read()) != -1) {
      System.out.print((char)i);
    }
    System.out.println();
  }
}
```

El objeto **entrada1** se construye utilizando el alfabeto completo en minúsculas, mientras que **entrada2** contiene sólo las cinco primeras letras. He aquí la salida:

```
entrada1 es:
abcdefghijklmnopqrstuvwxyz
entrada2 es:
abcde
```

CharArrayWriter

CharArrayWriter es una implementación de un flujo de salida que utiliza un arreglo como destino. **CharArrayWriter** tiene dos constructores, aquí mostrados:

CharArrayWriter( )
CharArrayWriter(int *numCars*)

En la primera forma, se crea un búfer con el tamaño por omisión. En la segunda, se crea un búfer de tamaño igual al indicado por *numCars*. El búfer se guarda en el campo **buf** de **CharArrayWriter**. El tamaño del búfer aumenta automáticamente si es necesario. El número de caracteres guardados en el búfer está contenido en el campo **count** de la propia clase **CharArrayWriter**. Tanto **buf** como **count** son campos protegidos.

El siguiente ejemplo ilustra **CharArrayWriter** rehaciendo el programa ejemplo mostrado antes para ilustrar **ByteArrayOutputStream**. Produce la misma salida que la versión anterior.


```
// Ejemplo de CharArrayWriter.
import java.io.*;

class CharArrayWriterDemo {
  public static void main(String args[]) throws IOException {
    CharArrayWriter f = new CharArrayWriter ();
    String s = "Esto debería terminar en el arreglo";
    char buf[] = new char [s.length()];

    s.getChars(0, s.length(), buf, 0);
    f.write(buf);
    System.out.println("Búfer como cadena");
    System.out.println(f.toString());
    System.out.println("Al arreglo");

    char c [] = f.toCharArray();
    for (int i=0; i<c.length; i++) {
      System.out.print(c[i]);
    }

    System.out.println("\n A un FileWriter()");
    FileWriter f2 = new FileWriter{"test.txt");
    f.writeTo(f2);
    f2.close();
    System.out.println("Reinicializando");
    f.reset();
    for (int i=0; i<3; i++)
      f.write('X');
    System.out.println(f.toString());
  }
}
```

BufferedReader

BufferedReader mejora el rendimiento dotando a la entrada de un búfer. Tiene dos constructores:

BufferedReader(Reader *flujoEntrada*)
BufferedReader(Reader *flujoEntrada*, int *tamañoBufer*)

La primera forma crea un flujo de caracteres con búfer utilizando un tamaño de búfer por omisión. En la segunda, el tamaño del búfer se pasa en el argumento *tamañoBufer*.

Al igual que en el caso del flujo orientado a bytes, dotar de un búfer a un flujo de entrada de caracteres también proporciona la base necesaria para permitir moverse hacia atrás en el flujo por dentro del búfer disponible. Para permitirlo, **BufferedReader** implementa los métodos **mark( )** y **reset( )**, y **BufferedReader.markSupported( )** devuelve verdadero.

El siguiente ejemplo rehace el ejemplo del **BufferedInputStream**, mostrado anteriormente, para que use un flujo de caracteres tipo **BufferedReader** en vez de un flujo de bytes con búfer. Como antes, se utilizan los métodos **mark( )** y **reset( )** para analizar un flujo buscando el símbolo de copyright en HTML. Esta etiqueta comienza con"&"y termina con punto y coma (;) sin que haya espacios en blanco de por medio. La entrada ejemplo tiene dos"&", para mostrar los casos en que se da el **reset( )** y en que no. La salida es la misma antes mostrada.

```
// Uso de entrada con búfer.
import java.io.*;

class BufferedReaderDemo {
  public static void main(String args[]) throws IOExeeption {
    String s = "Esto &copy; es un símbolo de copyright " +
      "pero esto &copy no lo es.\n";
    char buf[] = new char[s.length( )];
    s.getChars(0, s.length(), buf, 0);
    CharArrayReader in = new CharArrayReader(buf);
    BufferedReader f = new BufferedReader{in);
    int c;
    boolean marcado = false;

    while ((c = f.read()) != -1) {
      switch(c) {
      case '&':
        if (!marcado) {

          f.mark(32);
          marcado = true;
        } else {
          marcado = false;
        }
        break;
      case ';':
        if (marcado) {
          marcado = false;
          System.out.print("(c)");
        } else
          System.out.print((char) c);
        break;
      case ' ':
        if (marcado) {
        marcado = false;
        f.reset( );
        System.out.print("&");
      } else
        System.out.print((char) c);
      break;
      default:
```

```
        if (!marcado)
            System.out.print((char) c);
        break;
      }
    }
  }
}
```

BufferedWriter

Un **BufferedWriter** es un **Writer** que añade un búfer de salida. El uso de un **BufferedWriter** puede mejorar el rendimiento al reducir el número de veces que los datos se escriben físicamente en el flujo de salida

Un **BufferedWriter** tiene estos dos constructores:

BufferedWriter(Writer *flujoSalida*)
BufferedWriter(Writer *flujoSalida*, int *tamañoBufer*)

La primera forma crea un flujo con búfer utilizando un búfer con el tamaño por omisión. En la segunda, el tamaño del búfer se pasa en *tamañoBufer*.

PushbackReader

La clase **PushbackReader** permite devolver uno o más caracteres al flujo de entrada. Esto permite mirar adelante en el flujo de entrada. Aquí están sus dos constructores:

PushbackReader(Reader *flujoEntrada*)
PushbackReader(Reader *flujoEntrada*, int *tamañoBufer*)

La primera forma crea un flujo con búfer que permite devolver un carácter. En la segunda, el tamaño del búfer de devolución se pasa en *tamañoBufer*.

PushbackReader proporciona el método **unread( )**, que devuelve uno o más caracteres al flujo de entrada que invoca. Tiene tres formas:

void unread(int *car*)
void unread(char *bufer*[ ])
void unread(char *bufer*[ ], int *desplazo*, int *numCars*)

La primera forma devuelve al flujo el carácter pasado en *car*. Éste será el siguiente carácter proporcionado por una subsiguiente llamada a **read( )**. La segunda forma devuelve al flujo los caracteres en *bufer*. La tercera forma devuelve al flujo *numCars* caracteres comenzando en la *posición* dada por el parámetro desplazo. Si se intenta devolver un carácter cuando el búfer de devolución está lleno, se producirá una **IOException**.

El siguiente programa rehace el ejemplo anterior sobre **PushbackInputStream**, reemplazando **PushBackInputStream** por un **PushbackReader**. Como antes, el ejemplo ilustra cómo un analizador de un lenguaje de programación puede utilizar un flujo con devolución para manejar la diferencia entre el operador de comparación = = y el operador de asignación =.

```
// Ejemplo de unread( ).
import java.io.*;

class PushbackReaderDemo {
  public static void main(String args[]) throws IOExeeption {
```

```
    String s = "if (a = = 4) a = 0;\n";
    char buf [] = new char [s.length() ];
    s.getChars(0, s.length(), buf, 0);
    CharArrayReader in = new CharArrayReader(buf);
    PushbackReader f = new PushbackReader(in);
    int c;

    while ((c = f.read()) != -1) {
      switch (c) {
      case '=':
        if ((e = f.read( )) == '=')
          System.out.print(".eq.");.
        else {
          System.out.print("<-");
          f.unread(c);
        }
        break;
    default:
      System.out.print((char) c);
      break;
      }
    }
  }
}
```

PrintWriter

PrintWriter es esencialmente una versión orientada a caracteres de **PrintStream**. Implementa las interfaces **Appendable**, **Closeable** y **Flushable**. **PrintWriter** tiene muchos constructores. Los siguientes han sido proporcionados por **PrintWriter** desde el principio:

PrintWriter(OutputStream *flujoSalida*)
PrintWriter(OutputStream *flujoSalida*, boolean *vaciarSiNuevaLínea*)

PrintWriter(Writer *flujoSalida*)
PrintWriter(Writer *flujoSalida*, boolean *vaciarSiNuevaLínea*)

Donde *flujoSalida* especifica un **OutputStream** abierto que recibirá la salida. El parámetro *vaciarSiNuevaLínea* controla si se vacía automáticamente el flujo de salida cada vez que **println( )**, **printf( )** o **format( )** son llamados. Si *vaciarSiNuevaLínea* es **verdadero**, el vaciado tiene lugar automáticamente. Si es **falso**, el vaciado no es automático. Los constructores primero y tercero no vacían automáticamente.

El siguiente conjunto de constructores proporcionan una forma fácil de construir un **PrintWriter** que escribe su salida en un archivo.

PrintWriter(File *archSalida*) throws FileNotFoundException
PrintWriter(File *archSalida*, String *ConjuntoCars*)
  throws FileNotFoundException, UnsupportedEncodingException

PrintWriter(File *nomarchSalida*) throws FileNotFoundException
PrintWriter(File *nomarchSalida*, String *ConjuntoCars*)
  throws FileNotFoundException, UnsupportedEncodingException

Estos constructores permiten a **PrintWriter** ser creado desde un objeto **File** o especificando el nombre de un archivo. En cualquier caso, el archivo es automáticamente creado. Cualquier archivo preexistente con el mismo nombre será destruido. Una vez creado, el objeto **PrintWriter** dirige la salida al archivo especificado. Es posible especificar un carácter de codificación pasando su nombre en el parámetro *charSet*.

PrintWriter soporta los métodos **print( )** y **println( )** para todos los tipos de datos, incluidos los **Object**. Si un argumento no es de tipo primitivo, los métodos de **PrintWriter** llamarán al método **toString( )** para generar la salida correspondiente.

PrintWriter también soporta el método **printf( )**. Funciona de la misma forma que en la clase **PrintStream** descrita anteriormente: permite especificar el formato preciso de los datos. A continuación se muestra como **printf( )** está declarado en **PrinterWriter**:

PrintWriter printf(String *fortmatoString*, Object … *args*)

PrintWriter printf(Locale, *loc*, String *formatoString*, Object … *args*)

La primera versión escribe *args* a la salida estándar en el formato especificado en *formatoString*, utilizando el locale por omisión. El segundo permite especificar un locale. Ambos devuelven el *PrintWriter* que invoca.

El método **format( )** es soportado también y tiene las siguientes formas generales:

PrinterWriter format(String *formatoString*, Object … *args*)

PrinterWriter format(Locale *loc*, String *formatoString*, Object … *args*)

Y funciona exactamente como **printf( )**.

La clase Console

Java SE 6 agrega la clase **Console**. La cual es utilizada para leer y escribir en la consola, si existe alguna. Esta clase implementa la interfaz **Flushable**. **Console** es ante todo una clase que brinda comodidad porque la mayor parte de su funcionalidad está disponible a través de **System.in** y **System.out**. Sin embargo, su uso puede simplificar algunos tipos de interacción con consola, especialmente cuando se leen cadenas de la consola.

Console no provee constructores. En su lugar, un objeto **Console** se obtiene llamando a **System.console( )**, el cual está definido como:

static Console console( )

Si una consola está disponible, entonces devuelve una referencia a ella. De otra forma, se devuelve **null**. Una consola podrá no estar disponible en todos los casos. Así, si se devuelve **null**, no existe una consola de E/S.

Console define los métodos que se muestran en la Tabla 19-5. Note que los métodos de entrada, tales como **readLine( )**, generan una excepción de tipo **IOError** si ocurre un error de entrada. **IOError** es una nueva excepción agregada por Java SE 6, y es una subclase de Error. Esta excepción indica una falla de E/S que está fuera de control del programa. Por ello, normalmente no se gestiona una excepción **IOError**. Francamente, si un **IOError** es generado mientras se ingresa a la consola, usualmente significa que hay un catastrófico error de sistema.

También obsérvense los método **readPassword( )**. Estos métodos permiten a las aplicaciones leer una clave sin que aparezca en pantalla lo que se escribe. Cuando se leen claves no se debe generar salida alguna. Esto reduce las oportunidades de los programas maliciosos que serían capaces de obtener una clave escudriñando en la memoria.

Método	Descripción
void flush( )	Provoca que el contenido del búfer sea escrito físicamente en la consola.
Console format(String *formatoString*, Object … *args*)	Escribe *args* en la consola utilizando el formato especificado en *formatoString*.
Console printf(String *formatoString*, Object … *args*)	Escribe *args* en la consola utilizando el formato especificado en *formatoString*.
Reader reader( )	Devuelve la referencia al **Reader** conectado a la consola
String readLine( )	Lee y devuelve un cadena ingresada por el teclado. La entrada se detiene cuando el usuario presiona ENTER. Si el final del flujo de entrada de la consola se ha alcanzado se devuelve **null**. Una excepción **IOError** es lanzada en caso de falla.
String readLine(String *formatoString*, Object … *args*)	Muestra una cadena indicadora con el formato especificado en *formatoString* y *args*, luego lee y regresa un cadena ingresada mediante el teclado. La entrada se detiene cuando el usuario presiona ENTER. Si el final del flujo de entrada de la consola se ha alcanzado se devuelve **null**. Una excepción **IOError** es lanzada en caso de falla.
char[ ] readPassword( )	Lee una cadena ingresada mediante el teclado. La entrada se detiene cuando el usuario presiona ENTER. La cadena no se muestra. Si el final del flujo de entrada de la consola se ha alcanzado se devuelve **null**. Una excepción **IOError** es lanzada en caso de falla.
char[ ] readPassword(String *formatoString*, Object … *args*)	Muestra una cadena indicadora con el formato especificado en *formatoString* y *args*, luego lee una cadena ingresada mediante el teclado. La entrada se detiene cuando el usuario presiona ENTER. La cadena no se muestra. Si el final del flujo de entrada de la consola es alcanzado se devuelve **null**. Una excepción **IOError** es lanzada en caso de falla.
PrinterWriter writer( )	Devuelve una referencia al **Writer** conectado a la consola

TABLA 19-5 Métodos definidos por **Console**

A continuación está un ejemplo de la clase **Console**:

```
// Ejemplo de la clase Console.
import java.io.*;

class ConsoleDemo {
  public static void main(String args[]) {
    String str;
    Console con;
```

```
    // Obtiene una referencia a la consola.
    con = System.console();

    // Si no hay consola disponible, salir
    if (con == null) return;

    // Lee una cadena y luego la imprime.
    str = con.readLine("Ingresar una cadena: ");
    con.printf("Aquí está tú cadena: %s\n", str);
  }
}
```

Ésta es la salida del ejemplo:

```
Ingresar una cadena: Esta es una prueba
Aquí está tú cadena: Esta es una prueba
```


Uso de flujos de E/S

El siguiente ejemplo ilustra varias clases y métodos de flujos de caracteres de **E/S** en Java. Este programa implementa el comando estándar **wc** (word count). El programa tiene dos modos: si no se proporcionan nombres de archivo como argumentos, el programa opera sobre el flujo de entrada estándar. Si se especifica uno o más nombres de archivo, el programa opera sobre cada uno de ellos.

```
// Un programa para contar palabras.
import java.io.*;

class ContarPalabra {
  public static int palabras = 0;
  public static int lineas = 0;
  public static int caracteres = 0;

  public static void wc(InputStreamReader isr)
    throws IOException {
    int c = 0;
    boolean ultimoBlanco = true;
    String espacioBlanco = " \t\n\r";

    while ((c = isr.read()) != -1) {
      // Contar caracteres
      caracteres++;
      // Contar líneas
      if (c == '\n'){
        líneas++;
      }
      // Contar palabras detectando el comienzo de una palabra
      int index = espacioBlanco.indexOf(c);
      if (index == -1) {
        if (ultimoBlanco == true) {
        ++palabras;
        }
        ultimoBlanco = false;
      }
      else {
        ultimoBlanco = true;
```

```
      }
    }
    if(caracteres != 0) {
      ++ lineas;
    }
  }

  public static void main(String args[]) {
    FileReader fr;
    try {
      if (args.length == 0) { // Trabajando con entrada estándar stdin
        wc (new InputStreamReader(System.in));
      }
      else { // Trabajando con una lista de archivos
        for (int i = 0; i < args.length; i++) {
          fr = new FileReader(args[i]);
          wc(fr);
        }
      }
    }
    catch (IOException e) {
      return;
    }
    System.out.println(lineas + " " + palabras + " " + caracteres);
  }
}
```

El método **wc( )** opera sobre cualquier flujo de entrada y cuenta el número de caracteres, líneas y palabras. Hace un seguimiento de la paridad de las palabras y espacios en blanco con la variable **ultimoBlanco**.

Cuando se ejecuta sin argumentos, **ContarPalabra** crea un objeto **InputStreamReader** utilizando **System.in** como fuente del flujo. Este flujo se pasa entonces a **wc( )**, que hace el conteo. Cuando se ejecuta con uno o más argumentos, **ContarPalabra** supone que son nombres de archivos y crea objetos **FileReader** para cada uno de ellos, y los pasa al método **wc( )**. En cualquiera de los casos, imprime los resultados antes de terminar.

Mejora de wc( ) mediante la clase StreamTokenizer

Un modo todavía mejor de buscar patrones en un flujo de entrada es utilizar otra clase de E/S de Java: **StreamTokenizer**. Similar al **StringTokenizer** del Capítulo 18, **StreamTokenizer** divide el flujo de entrada en *tokens* que están delimitados por conjuntos de caracteres. Tiene este constructor:

StreamTokenizer (Reader *flujoEntrada*)

Aquí *flujoEntrada* debe de ser alguna forma de **Reader**.

StreamTokenizer define varios métodos. En este ejemplo utilizaremos sólo unos pocos. Para reiniciar el conjunto de delimitadores por omisión, utilizaremos el método **resetSyntax( )**. El conjunto de delimitadores por omisión está finamente ajustado para la separación en *tokens* de los programas Java y, por tanto adecuado para este ejemplo. Para nuestro ejemplo, nuestros tokens o "palabras", serán cualquier cadena de caracteres visibles consecutivos delimitados a ambos lados por un espacios en blanco.

Utilizamos el método **eolIsSignificant( )** para asegurar que los caracteres de la nueva línea se envían como tokens, de modo que podamos contar el número de líneas al igual que el de palabras. Ese método tiene la forma general:

void eolIsSignificant(boolean *banderaEOL*)

Si *banderaEOL* es **verdadera**, los caracteres fin de línea se devuelven como tokens. Si *banderaEOL* es **falsa**, los caracteres de fin de línea son ignorados.

El método **wordChars( )** se utiliza para especificar el rango de caracteres que se pueden utilizar en palabras. Su forma general se muestra aquí:

void wordChars(int *inicio*, int *final*)

Aquí, *inicio* y *final* indican el rango de caracteres válidos. En este programa, los caracteres en el rango 33 a 255 son caracteres de palabra válidos.

Los caracteres de espacio en blanco se especifican utilizando **whitespaceChars( )**. El método tiene esta forma general:

void whitespaceChars(int *inicio*, int *final*)

Aquí, *inicio* y *final* especifican el rango de caracteres de espacio en blanco válidos. El siguiente token se obtiene del flujo de entrada llamando a **nextToken( )**. Devuelve el tipo del token.

StreamTokenizer define cuatro constantes de tipo **int**: **TT_EOF**, **TT_EOL**, **TT NUMBER** y **TT_ WORD**. Existen tres variables de instancia**. nval** es una variable **double** pública utilizada para contener el valor de las palabras según se van reconociendo. **sval** es una variable pública de tipo **String** utilizada para mantener el valor de cualquier palabra conforme se van reconociendo. **ttype** es una variable de tipo **int** pública que indica el tipo de token que acaba de ser leído por el método **nextToken( )**. Si el token es una palabra, **ttype** es igual a **TT_WORD**. Si es un número, **ttype** equivale **TT_NUMBER**. Si el token es un carácter, **ttype** contiene su valor. Si se ha encontrado una condición de fin de línea, **ttype** es igual a **TT_EOL**, (esto presupone que **eolIsSignificant( )** ha sido llamado con un argumento **true**.) Si se ha alcanzado el final del flujo, **ttype** equivale **TT_EOF**.

El programa de conteo de palabras implementado con **StreamTokenizer** se muestra a continuación:

```
// Programa mejorado para contar palabras que utiliza StreamTokenizer
import java.io.*;

class ContarPalabra {
  public static int palabras=0;
  public static int lineas=0;
  public static int caracteres=0;

  public static void wc(Reader r) throws IOException {
    StreamTokenizer tok = new StreamTokenizer(r);

    tok.resetSyntax( );
    tok.wordChars(33, 255);
    tok.whitespaceChars(0, ' ');
    tok.eolIsSignificant(true);

    while (tok.nextToken( ) != tok.TT_EOF) {
      switch (tok.ttype) {
```

```
        case StreamTokenizer.TT_EOL:
          lineas++;
          caracteres++;
          break;
        case StreamTokenizer.TT_WORD:
          palabras++;
          default: // por omisión
            caracteres += tok.sval.length():
            break;
        }
      }
    }

    public static void main(String args[]) {
      if (args.length == 0) { // Trabajando con entrada estándar stdin
        try {
          wc (new InputStreamReader(System.in));
          System.out.println(lineas + " " + palabras + " " + caracteres);
        } catch (IOException e) {};
      } else { // Trabajando con una lista de archivos
        int tpalabras = 0, tcaracteres = 0, tlineas = 0:
        for (int i=0; i<args.length: i++) {
          try {
            palabras = caracteres = lineas = 0;
            wc(new FileReader(args[i]));
            tpalabras += palabras:
            tcaracteres += caracteres:
            tlineas += lineas:
            System.out.println(args[i] + ": " +
              lineas + " " + palabras + " " + caracteres);
          } catch (IOException e) {
            System.out.println(args[i] + ": error.");
          }
        }
        System.out.println("total: " +
        tlineas + " " + tpalabras + " " + tcaracteres);
      }
    }
  }
```

Serialización

La serialización es el proceso de escribir los datos de un objeto en un flujo de bytes. La serialización es útil cuando se quiere salvar el estado del programa en un área de almacenamiento persistente, como un archivo. Más tarde se pueden recuperar estos objetos utilizando el proceso de deserialización.

La serialización también se necesita para implementar la Invocación Remota de Métodos (RMI, por sus siglas en inglés). RMI permite que un objeto Java que está en una máquina llame a un método de un objeto Java que está en otra máquina. Se puede suministrar un objeto como argumento a ese método remoto. La máquina remitente serializa el objeto y lo transmite. La máquina destinataria lo deserializa. (Hay más información sobre el RMI en el Capítulo 27.)

Supongamos que un objeto que hay que serializar tiene referencias a otros objetos que, a su vez, tienen referencias a otros objetos. Este conjunto de objetos y las relaciones entre ellos

forman un grafo dirigido. Puede haber también referencias circulares dentro de este grafo de objetos. Esto es, el objeto X puede contener una referencia al objeto Y, y el objeto Y contener una referencia de vuelta al objeto X. Los objetos también pueden contener referencias a sí mismos. Las facilidades de serialización y deserialización de objetos se han diseñado para trabajar correctamente en estos escenarios. Si se intenta serializar un objeto en la cúspide de un grafo de objetos, todos los demás objetos referenciados son localizados y serializados recursivamente.

Análogamente, durante el proceso de deserialización, todos estos objetos y sus referencias se restauran correctamente.

A continuación daremos vistazo general al conjunto de las interfaces y clases que dan soporte a la serialización.

Serializable

Sólo un objeto que implemente la interfaz **Serializable** puede salvarse y restaurarse mediante las facilidades de serialización. La interfaz **Serializable** no define ningún miembro. Simplemente se utiliza para indicar que una clase se puede serializar. Si una clase es serializable, todas sus subclases lo son también.

Las variables declaradas con el modificador **transient** no son almacenadas por las facilidades de serialización. Igualmente, las variables static tampoco se almacenan.

Externalizable

Las facilidades de Java para la serialización y deserialización se han diseñado de modo que gran parte del trabajo de salvar y restaurar el estado de un objeto ocurra automáticamente. Sin embargo, hay casos en que el programador puede necesitar tener control sobre estos procesos. Por ejemplo, puede ser deseable utilizar técnicas de compresión o encriptación. La interfaz **Externalizable** está diseñada para estas situaciones.

La interfaz **Externalizable** define estos dos métodos:

void readExternal(ObjectInput *flujoEntrada*)
  throws IOException, *ClassNotFoundException*

void writeExternal(ObjectOutput *flujoSalida*)
  throws IOException

En estos métodos, *flujoEntrada* es el flujo de bytes del que se debe leer el objeto, y *flujoSalida* es el flujo de bytes en el que hay que escribir el objeto.

ObjectOutput

La interfaz **ObjectOutput** extiende la interfaz **DataOutput** y soporta la serialización de objetos. En la Tabla 19-6 se muestran los métodos definidos por esta clase. Obsérvese especialmente el método **writeObject( )**. A esto se le llama serializar un objeto. Todos estos métodos producen una **IOException** bajo condiciones de error.

ObjectOutputStream

La clase **ObjectOutputStream** extiende la clase **OutputStream** e implementa la interfaz **ObjectOutput**. Su función es escribir objetos a un flujo. El constructor de esta clase es:

ObjectOutputStream(OutputStream *flujoSalida*) throws IOException

El argumento *flujoSalida* es el flujo de salida en el que se escribirán los objetos serializados.

Los métodos más comúnmente utilizados en esta clase se muestran en la Tabla 19-7. Estos producen una **IOException** bajo condiciones de error. Existe también una clase interior a **ObjectOuputStream** llamada **PutField**. Esta clase facilita la escritura de campos persistentes y su uso se sale del ámbito de este libro.

Método	Descripción
void close( )	Cierra el flujo que invoca. Intentos posteriores de escritura generarán una **IOException**.
void flush( )	Finaliza el estado de la salida de modo que los búferes se limpien. Esto es, vacía los búferes de salida.
void write(byte *bufer*[ ])	Escribe un arreglo de bytes al flujo que invoca.
void write(byte *bufer*[ ], int *desplazo*, int *numBytes*)	Escribe un subrango de *numBytes* bytes del arreglo *bufer*, comenzando por *bufer*[*desplazo*].
void write(int *b*)	Escribe un byte único en el flujo que invoca. El byte escrito es el byte menos significativo de *b*.
void writeObject(Object *obj*)	Escribe el objeto *obj* al flujo que invoca

TABLA 19-6 Los métodos definidos por **ObjectOutput**

Método	Descripción
void close( )	Cierra el flujo que invoca. Intentos posteriores de escritura generarán una **IOException**.
void flush( )	Finaliza el estado de la salida de modo que se limpien los búferes. Esto es, vacía los búferes de salida.
void write(byte *bufer*[ ])	Escribe un arreglo de bytes al flujo que invoca.
void write(byte *bufer*[ ], int *desplazo*, int *numBytes*)	Escribe un subintervalo de *numBytes* bytes del arreglo *bufer* comenzando por *bufer*[*desplazo*].
void write(int *b*)	Escribe un **byte** único al flujo que invoca. El byte escrito es el byte menos significativo de *b*.
void writeBoolean(boolean *b*)	Escribe un valor de tipo **boolean** al flujo que invoca.
void writeByte(int *b*)	Escribe un **byte** al flujo que invoca. El byte escrito es el byte menos significativo de *b*.
void writeBytes(String *cad*)	Escribe los bytes que representan a *cad* en el flujo que invoca.
void writeChar(int *c*)	Escribe un **char** al flujo que invoca.
void writeChars(String *cad*)	Escribe los caracteres en **cad** al flujo que invoca
void writeDouble(double *d*)	Escribe un **double** al flujo que invoca.
void writeFloat(float *f*)	Escribe un **float** al flujo que invoca.
void writeInt(int *i*)	Escribe un **int** al flujo que invoca.
void writeLong(long *l*)	Escribe un **long** al flujo que invoca.
final void write Object(Object *obj*)	Escribe **obj** al flujo que invoca.
void writeShort(int *i*)	Escribe un **short** al flujo que invoca.

TABLA 19-7 Los métodos definidos por **ObjectOutputStream**

ObjectInput

La interfaz **ObjectInput** extiende la interfaz **DataInput** y define los métodos mostrados en la Tabla 19-8. Ésta soporta la serialización de objetos. Nótese especialmente el método **readObject( )**. A este se le llama para deserializar un objeto. Todos estos métodos producirán una **IOException** en condiciones de error. El método **readObject( )** puede también generar una excepción de tipo **ClassNotFoundException**.

ObjectInputStream

La clase **ObjectInputStream** extiende de la clase **InputStream** e implementa la interfaz **ObjectInput**. **ObjectInputStream** se ocupa de la lectura de objetos desde un flujo. El constructor de esta clase es:

ObjectInputStream(InputStream *flujoEntrada*)
throws IOException

El argumento *flujoEntrada* es el flujo de entrada del que se leen los objetos serializados.

Los métodos más comúnmente utilizados en esta clase podemos verlos en la Tabla 19-9. Esto métodos producen una **IOException** en condiciones de error. El método **readObject( )** puede también lanzar una excepción de tipo **ClassNotFoundException**. Existe también una clase interna a **ObjectInputStream** llamada **GetField**. Esta clase facilita la lectura de campos persistentes y su uso se sale del ámbito de este libro.

Un ejemplo de serialización

El siguiente programa ilustra cómo utilizar la serialización y deserialización de objetos. Comienza instanciando un objeto de clase **MiClase**. Este objeto tiene tres variables de instancia que son de los tipos **String**, **int** y **double**. Ésta es la información que queremos salvar y recuperar.

Método	Descripción
int available( )	Devuelve el número de bytes actualmente disponibles en el bufer de entrada.
void close( )	Cierra el flujo que invoca. Intentos de lectura posteriores generarán una **IOException**.
int read( )	Devuelve una representación como entero del siguiente byte de entrada disponible. Se devuelve –1 cuando se ha alcanzado el final del archivo.
int read(byte *bufer*[ ])	Intenta leer hasta *bufer.length* bytes de *bufer*, devolviendo el número de bytes leídos con éxito. Se devuelve –1 cuando se ha alcanzado el final del archivo
int read(byte *bufer*[ ], int *desplazo*, int *numBytes*)	Intenta leer hasta *numBytes* bytes de *bufer* comenzando con *bufer*[*desplazo*], devolviendo el número de bytes que fueron leídos con éxito. Se devuelve –1 cuando se ha alcanzado el final del archivo.
Object readObject( )	Lee un objeto del flujo que invoca.
long skip(long *numBytes*)	Ignora (esto es, salta) *numBytes* bytes del flujo que invoca, devolviendo el número de bytes realmente ignorados.

TABLA 19-8 Los métodos definidos por **ObjectInput**

Método	Descripción
int available( )	Devuelve el número de bytes actualmente disponibles en el búfer de entrada.
void close( )	Cierra el flujo que invoca. Intentos posteriores de lectura generarán una **IOException**.
int read( )	Devuelve una representación como entero del siguiente byte de entrada disponible. Se devuelve –1 si se ha alcanzado el final del archivo.
int read(byte *bufer*[ ], int *desplazo*, int *numBytes*)	Intenta leer hasta *numBytes* bytes de *bufer* comenzando en *bufer*[*desplazo*], devolviendo el número de bytes leídos con éxito. Se devuelve –1 si se ha alcanzado el final del archivo.
boolean readBoolean( )	Lee y devuelve un **boolean** del flujo que invoca.
byte readByte( )	Lee y devuelve un **byte** del flujo que invoca.
char readChar( )	Lee y devuelve un **char** del flujo que invoca.
double readDouble( )	Lee y devuelve un **double** del flujo que invoca.
float readFloat( )	Lee y devuelve un **float** del flujo que invoca.
void readFully(byte *bufer*[ ])	Lee *bufer.length* bytes de *bufer*. Vuelve sólo cuando todos los bytes han sido leídos.
void readFully(byte *bufer*[ ],int *desplazo*, int *numBytes*)	Lee *numBytes* bytes de *bufer* comenzando en *bufer*[*desplazo*]. Vuelve sólo cuando se han leído numBytes
int readInt( )	Lee y devuelve un **int** del flujo que invoca.
long readLong( )	Lee y devuelve un **long** del flujo que invoca.
final Object readObject( )	Lee y devuelve un objeto del flujo que invoca.
short readShort( )	Lee y devuelve un **short** del flujo que invoca.
int readUnsignedByte( )	Lee y devuelve un **byte** sin signo del flujo que invoca.
int readUnsignedShort( )	Lee un **short** sin signo del flujo que invoca.

TABLA 19-9 Métodos de uso común definidos por **ObjectInputStream**

Se crea un **FileOutputStream** que se refiere a un archivo llamado"serial"y se crea un **ObjectOutputStream** para ese flujo de salida a archivo. Luego, el método **writeObject( )** de **ObjectOutputStream** se utiliza para serializar nuestro objeto. El flujo de salida de objetos se vacía y se cierra.

A continuación se crea un **FileInputStream** que se refiere al archivo llamado"serial", y se crea un **ObjectInputStream** para ese flujo de entrada desde archivo. El método **readObject( )** de **ObjectInputStream** se utiliza entonces para deserializar nuestro objeto. Finalmente, se cierra el flujo de entrada de objetos.

Nótese que **MiClase** se define de modo que implemente la interfaz **Serializable**. Si esto no se hace, se produce una excepción de tipo **NotSerializableException**. Experimentemos con este programa declarando como **transient** algunas de las variables de instancia de **MiClase**. Eso ocasionará que esos datos no se almacenen en la serialización.

```
import java.io.*;

public class SerializationDemo {
  public static void main{String args[]) {

    // Serialización de objeto
    try {
      MiClase objetol = new MiClase ("Hola", -7, 2.7el0);
      System.out.println("objetol: " + objetol);
      FileOutputStream fos = new FileOutputStream("serial");
      ObjectOUtputStream oos = new ObjectOUtputStream(fos);
      oos.writeObject(objetol);
      oos.flush();
      oos.close();
    }
    catch(Exception e) {
      System.out.println("Excepción durante la serialización: " + e);
      System.exit(0);
    }

    // Deserialización de objeto
    try {
      MiClase objet02;
      FileInputStream fis = new FileInputStream("serial");
      ObjectInputStream ois = new ObjectInputStream(fis);
      objet02 = (MiClase)ois.readObject();
      ois.close();
      System.out.println("objet02: " + objet02);
    }
    catch(Exception e) {
      System.out.println("Excepción durante la serialización: " + e);
      System.exit(0);
    }
  }
}

class MiClase implements Serializable {
  String s;
  int i;
  double d;
  public MiClase(String s, int i, double d) {
    this.s = s;
    this.i = i;
    this.d = d;
  }
  public String toString() {
    return "s=" + s + "; i=" + i + "; d=" + d;
  }
}
```

Este programa muestra que las variables de instancia de **objeto1** y **objeto2** son idénticas. La salida se muestra aquí:

```
objetol: s=Hola; i=-7; d=2.7El0
objeto2: s=Hola; i=-7; d=2.7El0
```

Ventajas de los flujos

La interfaz de flujos de E/S de Java proporciona una abstracción clara para una tarea compleja y a menudo engorrosa. La composición de las clases de flujos con filtro permite construir dinámicamente la interfaz de flujo personalizada que mejor se adapte a los requisitos de transferencia de datos. Los programas Java que se adhieran al uso de las clases abstractas de alto nivel **InputStream**, **OutputStream**, **Reader** y **Writer** funcionarán adecuadamente en el futuro incluso aunque se inventen clases de flujos concretas nuevas y mejoradas. Como veremos en el siguiente capítulo, este modelo funciona muy bien cuando pasamos de un conjunto de flujos basados en un sistema de archivos a los flujos de red y a los flujos con sockets. Finalmente, se espera que en el futuro la serialización de objetos desempeñe un papel cada vez más importante en la programación en Java. Las clases de E/S de serialización de Java proporcionan una solución portable a ésta, en ocasiones delicada, tarea.

20 CAPÍTULO

Trabajo en red

Como todos los lectores sabrán, Java es prácticamente un sinónimo de programación para Internet. Existen muchas razones para esto, no sólo la habilidad para generar código seguro, multiplataforma y portátil. Sin embargo, una de las razones más importantes para que Java sea un lenguaje estrella para la programación en Internet son las clases definidas en el paquete java.net. Éste provee una forma fácil de uso mediante el cual todos los programadores de todos los niveles pueden acceder a los recursos en red.

Este capítulo estudia el paquete **java.net.** Es importante enfatizar que el trabajo en red es un tópico bastante largo y en ocasiones complicado. No es posible discutir en este libro todas las capacidades contenidas en el paquete **java.net.** En su lugar, este capítulo se enfoca en muchas de sus clases e interfaces principales.

Fundamentos del trabajo en red

Antes de comenzar, sería muy útil revisar algunos conceptos y términos clave del trabajo en red. El núcleo del trabajo en red en Java es el soporte al concepto de *socket* (conector). Un socket identifica un extremo de una red. El paradigma de sockets surge en el sistema operativo UNIX Berkeley 4.2BSD liberado a principios de 1980, por ello, en ocasiones se utiliza el termino *Berkeley socket*. Los sockets son el principio del trabajo en red moderno porque un socket permite a una computadora servir a varios clientes al mismo tiempo, así como servir diferentes tipos de información. Esto se realiza a través del uso de *puertos*, los cuales son sockets etiquetados en una máquina en particular. Un proceso servidor se dice estar "escuchando" a un puerto hasta que un cliente se conecte a él. Un servidor permite aceptar múltiples clientes conectados al mismo número de puerto, aunque cada sesión es única. Para manejar múltiples conexiones de clientes, un proceso servidor debe ser multihilo o tener otras formas de multiplexar de forma simultánea las peticiones de E/S.

La comunicación de un socket se realiza vía un protocolo. El *Protocolo de Internet* (IP) es un protocolo de ruteo de bajo nivel que fracciona la información en pequeños paquetes y los envía a una dirección a través de la red, lo cual no garantiza la entrega de dichos paquetes a su destino. El *Protocolo de Control de Transmisión* (TPC) es un protocolo de alto nivel que maneja el encadenamiento de paquetes, clasificándolos y retransmitiéndolos tanto como se necesite para transmitir los datos de forma segura. Un tercer protocolo, el *Protocolo de Datagrama de Usuario* (UDP) se localiza al mismo nivel que TCP y puede ser utilizado para implementar un rápido, pero poco confiable, transporte de paquetes.

Una vez que la conexión se ha establecido, un protocolo de alto nivel se presenta; el protocolo depende del puerto que se esté usando. TCP/IP reserva los puertos menores a 1 024 para protocolos

específicos, muchos de los cuales le serán familiares, si ha navegado por Internet. El puerto número 21 es para FTP; el 23 es para Telnet; el 25 es para correo electrónico; el 43 es para whois; el 79 es para finger; el 80 es para HTTP; el 119 es para netnews y la lista continúa. Depende de cada protocolo determinar cómo un cliente deberá interactuar con el puerto.

Por ejemplo, HTTP es el protocolo que los navegadores y los servidores Web utilizan para transferir páginas de hipertexto e imágenes. Es un protocolo simple para un servicio de navegación Web simple. Aquí se muestra como funciona. Cuando un cliente solicita un archivo a un servidor HTTP, una acción conocida como *hit* (golpear), simplemente manda el nombre del archivo en formato especial a un puerto predefinido y lee de regreso el contenido del archivo. El servidor también responde con un código de estado para decirle al cliente si la solicitud puede ser completada o no y por qué.

Un componente clave de Internet es la *dirección*. Cada computadora sobre el Internet tiene una. Una dirección de Internet es un número que identifica de forma única a cada computadora en la red. Originalmente, todas las direcciones de Internet consistían en valores de 32 bits, organizados en bloques de 8 bits. Este tipo de dirección fue especificada por IPv4 (Protocolo de Internet, versión 4). Sin embargo, un nuevo esquema de direccionamiento, llamado IPv6 (Protocolo de Internet, versión 6) ha entrado a las redes. IPv6 utiliza un valor de 128 bits para representar una dirección, divido en ocho bloques de 16 bits cada uno. Aunque existen muchas razones para cambiar a IPv6, la principal es que soporta un rango más amplio de direcciones que el IPv4.

Para proveer compatibilidad con IPv4, los primeros 32 bits de una dirección IPv6 pueden contener una dirección IPv4 válida. Así, IPv4 es ascendentemente compatible con IPv6. Afortunadamente, cuando se utilice Java, normalmente no se necesitará preocuparse por utilizar direccionamiento IPv4 o IPv6, dado que Java gestiona estos detalles.

Tal como los números de una dirección IP describen una jerarquía de red, el nombre de una dirección de Internet, llamado *nombre de dominio*, describe la localidad de una máquina en el ciberespacio. Por ejemplo, www.mcgraw-hill-educacion.com está en el domino COM (reservado para sitos comerciales); es llamado McGraw-Hill educación (el nombre de la compañía), y www identifica el servidor para solicitudes Web. Un dominio de Internet es dirigido a una dirección IP mediante el *Servicio de Nombres de Dominio* (DNS). Esto permite a los usuarios trabajar con nombres de dominio pero el Internet trabaja con direcciones IP.

Las clases e interfaces para el trabajo en red

Java soporta TCP/IP tanto extendiendo la interfaz de E/S por flujos ya establecida, que introdujimos en el Capítulo 19, como añadiendo las características requeridas para construir objetos de E/S dentro de la red. Java soporta los protocolos TCP y UDP. TCP se usa para la E/S fiable basada en flujos dentro de la red. UDP soporta un modelo punto a punto orientado a datagramas; es más simple, y por tanto más rápido. Las clases contenidas en el paquete **java.net** son:

Authenticator	Inet6Address	ServerSocket
CacheRequest	InetAddress	Socket
CacheRequest	InetSocketAddress	SocketAddress
ContentHandler	InterfaceAddress (Agregado por Java SE 6)	SocketImpl
CokieHandler	JarURLConnection	SocketPermissions

CokieManager (Agregado por Jave SE 6)	MulticastSocket	URI
DatagramPacket	NetPermission	URL
DatagramSocket	NetworkInterface	URLClassLoader
DatagramSocketImpl	PasswordAuthentication	URLConnection
HttpCookie (Agregado por Jave SE 6)	Proxy	URLDecoder
HttpURLConnection	ProxySelector	URLEncoder
IDN (Agregado por Java SE 6)	ResponseCache	URLStreamHandler
Inet4Address	SecureCacheResponse	

El paquete **java.net** contiene las siguientes interfaces:

ContentHandlerFactory	DatagramSocketImplFactory	SocketOptions
CokiePolicy (Agregado por Java SE 6)	FileNameMap	URLStreamHandlerFactory
CokieStore (Agregado por Java SE 6)	SocketImplFactory	

En las siguientes secciones examinaremos las principales clases para trabajo en red y daremos varios ejemplos que las aplican. Una vez que entendamos el núcleo de estas clases para trabajo en red, seremos capaces de explorar fácilmente otras por cuenta propia.

InetAddress

La clase **InetAddress** se utiliza para encapsular tanto la dirección IP como el nombre de dominio de esa dirección. Se interactúa con esta clase utilizando el nombre de un nodo IP, que es más cómodo y comprensible que su dirección IP. La clase **InetAddress** oculta el número en su interior. **InetAddress** puede gestionar tanto direcciones IPv4 como IPv6.

Métodos de fábrica

La clase **InetAddress** no tiene constructores visibles. Para crear un objeto **InetAddress** hay que utilizar uno de los métodos de fábrica disponibles. Los *métodos de fábrica* son simplemente una convención por la cual los métodos estáticos de una clase devuelven una instancia de esa clase; en ocasiones hacer esto, en lugar de sobrecargar constructores con diferentes listas de parámetros, permite mayor claridad. Los tres métodos de fábrica más comúnmente utilizados se muestran a continuación:

static InetAddress getLocalHost( ) throws UnknownHostException

static InetAddress getByName(String *nomHost*) throws UnknownHostException

static InetAddress[ ] getAllByName(String *nomHost*) throws UnknownHostException

El método **getLocalHost( )** simplemente devuelve el objeto **InetAddress** que representa al nodo local. El método **getByName( )** devuelve un **InetAddress** del nombre de nodo que

se le pase. Si estos métodos son incapaces de resolver el nombre del nodo, producen una **UnknownHostException**.

En Internet es común que un único nombre represente varias máquinas. En el mundo de los servidores Web, éste es un modo de conseguir cierto grado de escalamiento en el uso de recursos. El método de fábrica **getAllByName( )** devuelve un arreglo de objetos **InetAddress** que representa a todas las direcciones en que se resuelve un nombre particular. También producirá una **UnknownHostException** si no puede resolver el nombre con al menos una dirección.

InetAddress también incluye al método de fábrica **getByAddress( )** el cual toma una dirección IP y devuelve un objeto **InetAddress**. Cualquier dirección IP ya sea IPv4 o IPv6 puede ser utilizada.

El siguiente ejemplo imprime las direcciones y nombres de la máquina local y de dos sitios Web conocidos en Internet:

```
// Ejemplo con InetAddress.
import java.net.*;

class InetAddressTest
{
  public static void main (String args[]) throws UnknownHostException {
    InetAddress Address = InetAddress.getLocalHost();
    System.out.println(Address);
    Address = InetAddress.getByName ("osborne.com");
    System.out.println(Address);
    InetAddress SW [] = InetAddress.getAllByName ("www.nba.com");
    for (int i=0; i<SW.length; i++)
      System.out.println(SW[i]);
  }
}
```

Aquí está la salida producida por este programa. Por supuesto, la salida en su computadora puede ser ligeramente distinta.

```
default/206.148.209.138
osbome.com/198.45.24.162
www.nba.com/64.5.96.214
www.nba.com/64.5.96.214
```

Métodos de Instancia

La clase **InetAddress** también tiene unos pocos métodos no estáticos, que se pueden utilizar sobre los objetos devueltos por los métodos que acabamos de ver, a continuación algunos de los más comúnmente utilizados:

boolean equals (Object *other*)	Devuelve **verdadero** si este objeto tiene la misma dirección de Internet que *otro*.
byte[ ] getAddress( )	Devuelve un arreglo de bytes que representa la dirección IP del objeto, los bytes en el orden comentado anteriormente.
String getHostAddress( )	Devuelve una cadena que representa la dirección del nodo asociado al objeto **InetAddress**
String getHostName( )	Devuelve una cadena que representa el nombre del nodo asociado al objeto **InetAddress**.

boolean isMulticastAddress( )	Devuelve **verdadero** si esta dirección de Internet es una dirección multicast. Si no, devuelve **falso**.
String toString( )	Devuelve una cadena que lista el nombre del nodo y la dirección IP.

Las direcciones de Internet se buscan en una serie de servidores organizados jerárquicamente. Esto significa que la computadora local podría conocer la equivalencia de un nombre a dirección IP automáticamente, por ejemplo la propia y la de los servidores cercanos. Para otros nombres, puede preguntarle a un servidor DNS local. Si ese servidor no tiene información para una dirección especifica, puede ir a un sitio remoto y preguntarla. Esto puede continuar subiendo hasta el servidor raíz. Este proceso puede llevar bastante tiempo, por lo que es bueno estructurar el código de modo que la información de direcciones IP se guarde localmente en memoria caché en vez de buscarla fuera repetidamente.

Inet4Address e Inet6Address

A partir de la versión 1.4, Java ha incluido soporte para direccionamiento IPv6. Gracias a esto, dos subclases de **InetAddress** fueron creadas: **Inet4Address** e **Inet6Address**. **Inet4Address** representa un estilo tradicional de direccionamiento para IPv4. **Inet6Address** encapsula un nuevo estilo de direccionamiento para IPv6. Dado que éstas son subclases de **InetAddress**, una referencia **InetAddress** puede hacer referencia a cualquiera. Ésta es la forma en que Java fue capaz de agregar funcionalidad IPv6 sin necesidad de frenar al código existente o de agregar muchas más clases. La mayoría de las veces, se puede simplemente utilizar **InetAddress** cuando se trabaja con direcciones IP debido a que esta clase se puede adaptar a ambos estilos.

Conectores TCP/IP para clientes

Los sockets (conectores) TCP/IP se utilizan para implementar conexiones fiables, bidireccionales, persistentes, punto a punto y basadas en flujos entre nodos en Internet. Un socket se puede utilizar para conectar el sistema de E/S de Java con otros programas que pueden residir en la máquina local o en cualquier otra máquina en Internet.

NOTA Los applets solo pueden establecer conexiones mediante sockets de vuelta al nodo desde el que se descargó el applet. Esta restricción existe porque sería peligroso que applets cargados a través de un firewall tuvieran acceso a cualquier máquina.

Hay dos clases de sockets TCP en Java. Uno es para servidores, y el otro es para clientes. La clase **ServerSocket** está diseñada para ser un "listener" (escucha), que espera a que se conecten clientes antes de hacer algo. Así, **ServerSocket** es para servidores. La clase **Socket** es para clientes y está diseñada para conectarse a **sockets** servidores e iniciar intercambios de información bajo un protocolo. Debido a que los **sockets** clientes son los más comúnmente utilizados por las aplicaciones de Java, se examinan a continuación.

La creación de un objeto **Socket** establece implícitamente una conexión entre cliente y servidor. No hay métodos o constructores que expongan explícitamente los detalles de cómo se establece la conexión. Aquí se muestran dos constructores utilizados para crear sockets cliente:

Socket(String *nomHost*, int *puerto*) throws UnknownHostException, IOException	Crea un socket que conecta el nodo local con el nodo y puerto dados.
Socket(InetAddress *direcciónIP*, int *puerto*) throws IOException	Crea un socket utilizando un objeto **InetAddress** preexistente y un puerto

Socket define varios métodos de instancia. Por ejemplo, un **Socket** puede ser examinado en cualquier momento para obtener información de la dirección y puerto asociado con él, utilizando los siguientes métodos:

InetAddress getInetAddress( )	Devuelve el **InetAddress** asociado con el objeto Socket. Devuelve null si el socket no está conectado.
int getPort( )	Devuelve el puerto remoto al que este objeto **Socket** está conectado. Devuelve 0 si el socket no está conectado.
int getLocalPort( )	Devuelve el puerto local al que este objeto **Socket** está enlazado. Devuelve –1 si el socket no está enlazado.

Se puede obtener acceso a los flujos de entrada y salida asociados con un **Socket** mediante el uso de los métodos **getInputStream( )** y **getOutputStream( )** como se muestra a continuación. Cada uno puede lanzar una **IOException** si el socket ha sido invalidado por una pérdida de conexión. Estos flujos son utilizados exactamente igual que los flujos de E/S descritos en el capítulo 19 para enviar y recibir datos.

InputStream getInputStream( ) throws IOException	Devuelve el **InputStream** asociado al socket que invoca.
OutputStream getOutputStream( ) throws IOException	Devuelve el **OutputStream** asociado al socket que invoca.

Muchos otros métodos están disponibles incluyendo **connect( )**, el cual permite especificar una nueva conexión; **isConnected( )**, el cual devuelve verdadero si el socket está conectado a un servidor; **isBound( )**, el cuál devuelve verdadero si el socket está ligado a una dirección; e **isClosed( )**, el cual devuelve verdadero si el socket está cerrado.

El siguiente programa provee un ejemplo simple del uso de la clase **Socket**. Abre una conexión a un puerto "whois" (puerto 43) en el servidor InterNIC, envía el argumento de la línea de comandos por el socket, y luego imprime el dato devuelto. InterNIC intentará buscar el argumento como un nombre de dominio de Internet registrado, para luego devolver la dirección IP y la información de contacto para ese sitio.

```
//Ejemplo con Sockets.
import java.net.*;
import java.io.*;

class Whois {
  public static void main (String args []) throws Exception {
    int c;

    // Crea un socket y lo conecta a internic.net, puerto 43
    Socket s = new Socket("internic.net", 43);

    // Obtiene flujos de entrada y salida
```

```
    InputStream in = s.getInputStream();
    OutputStream out = s.getOutputStream();

    // Construye una cadena para la solicitud
    String str = (args.length == 0 ? "osbome.com" : args[0]) + "\n";

    // Convierte a bytes
    byte buf[] = str.getBytes();

    // Envía la solicitud
    out.write(buf);

    // Lee y muestra la respuesta
    while ((c = in.read()) != -1) {
    System.out.print((char) c);
    }
    s.close();
  }
}
```


Si, por ejemplo, se ha introducido **osborne.com** en la línea de comandos, se obtendría algo parecido a lo siguiente:

```
Whois Server Version 1.3

Domain names in the .com, .net, and .org domains can now be registered with
many different competing registrars. Go to http://www.internic.net for detai-
led information.

  Domain Name: OSBORNE.COM
  Registrar: NETWORK SOLUTIONS, INC.
  Whois Server: whois.networksolutions.com
  Referra1 URL: www.networkso1utions.com
  Name Server: NS1.EPPG.COM
  Name Server: NS2.EPPG.COM
.
.
.
```

A continuación se explica cómo es que el programa funciona. Primero, se construye un **Socket** y se especifica el nombre del sitio "internic.net"y el puerto número 43. **Internic.net** es el sitio web InterNIC que gestiona las solicitudes whois. El puerto 43 es el puerto para las peticiones whois. Luego, ambos flujos de entrada y salida son abiertos en el socket. Luego, se construye una cadena que contiene el nombre del sitio web del que se solicita información. En este caso, si el sitio web no se especifica en la línea de comando, entonces"osborne.com"se utiliza. La cadena se convierte en un arreglo de **bytes** y se envía al socket. La respuesta se lee por la entrada del socket; finalmente se muestra el resultado.

URL

El ejemplo anterior era poco claro, porque el interés del Internet moderno no está en los protocolos antiguos, como whois, finger y FTP. Está en la WWW, (World Wide Web) o Red Mundial. La Red (Web) es una colección holgada de protocolos de alto nivel y de formatos de archivos, todo ello unificado en un navegador web. Uno de los aspectos más importantes de la

red es que Tim Berners-Lee ideó una forma escalable de localizar todos los recursos en la red. Una vez que es posible nombrar fiablemente todas y cada una de las cosas, se convierte en un paradigma muy potente. El Localizador de Recursos Uniforme (URL) hace justamente eso.

El URL proporciona una forma razonablemente inteligible de identificar o direccionar unívocamente información en Internet. Los URL son ubicuos: todo navegador los utiliza para identificar información en la red. Dentro de la biblioteca de clases de red de Java, la clase **URL** proporciona una API simple y concisa para acceder a información a través de Internet utilizando URL.

Todos los URLs comparten el mismo formato, aunque están permitidas algunas variaciones. Dos ejemplos de URL son **http://www.osborne.com/** y **http://www.osborne.com:80/index.htm**. Una especificación URL se basa en cuatro componentes. El primero es el protocolo a utilizar, separada del resto del localizador por dos puntos (:). Protocolos comunes son HTTP, FTP, gopher y file, aunque hoy en día casi todo se hace vía HTTP (de hecho, la mayoría de los navegadores operarán correctamente si se omite el"http:// "de la especificación URL). El segundo componente es el nombre del nodo o dirección IP del nodo a utilizar; esto viene delimitado por la izquierda por dos barras (//), y por la derecha, por una barra (/), u opcionalmente dos puntos (:). El tercer componente, el número de puerto, es un parámetro opcional, delimitado por la izquierda del nombre del nodo mediante dos puntos (:) y por la derecha mediante una barra(/). (Por omisión el puerto 80 es el puerto HTIP predefinido, de modo que":80"es redundante). La cuarta parte es el nombre real del archivo con su directorio. La mayoría de los servidores HTTP añadirán un archivo llamado **index.html** o **index.htm** a un URL que se refiere directamente a un directorio. Por tanto, **http://www.osborne.com/**es lo mismo que **http://www.osborne.com/index.htm.**

La clase **URL** de Java tiene varios constructores, todos los cuales pueden producir una excepción **MalformedURLException**. Una forma comúnmente utilizada para especifica el URL es con una cadena idéntica a lo que un navegador muestra en pantalla:

URL(String *especificadorUrl*) throws MalformedURLException

Las siguientes dos formas del constructor permiten dividir el URL en sus partes componentes:

URL(String *nomProtocolo*, String nomNodo, int puerto, String rutaArchivo)

throws MalformedURLException

URL(String *nomProtocolo*, String *nomNodo*, String *rutaArchivo*)

throws MalformedURLException

Otro constructor frecuentemente usado permite utilizar un URL existente como contexto de referencia y crear un URL nuevo a partir de él. Aunque esto suena un poco retorcido, es en realidad bastante fácil y útil.

URL(URL *objUrl*, String *especificadorUrl*) throws MalformedURLException

En el siguiente ejemplo se crea un URL a la página de descargas de Osborne y a continuación se examinan sus propiedades:

```
// Ejemplo de URL.
import java.net.*;
class URLDemo {
```

```
  public static void main(String args[]) throws MalformedURLException {
    URL hp = new URL("http://www.osbome.com/download");

    System.out.println("Protocolo: " + hp.getProtocol());
    System.out.println("Puerto: " + hp.getport());
    System.out.println("Nodo: " + hp.getHost());
    System.out.println("Archivo: " + hp.getFile());
    System.out.println("Extensión: " + hp.toExtemalForm());
  }
}
```

Al correr esto, se obtiene la siguiente salida:

```
Protocolo: http
Puerto: -1
Nodo: www.osbome.com
Archivo: /download
Extensión: http://www.osbome.com/download
```

Note que el puerto es –1; esto significa que no se ha establecido explícitamente uno. Ahora que hemos creado un objeto **URL**, queremos obtener los datos asociados a él. Para acceder realmente a los bits o al contenido de información de un objeto **URL** se crea, a partir de él, un objeto **URLConnection** utilizando su método **openConnection( )**, de la siguiente forma:

```
urlc = url.openConnection()
```

openConnection( ) tiene la siguiente forma general:

URLConnection openConnection( ) throws IOException

Devuelve un objeto **URLConnection** asociado con el objeto **URL** que invoca. Puede producir una **IOException**.

URLConnection

URLConnection es una clase de propósito general para acceder a los atributos de una fuente remota. Una vez establecida una conexión a un servidor remoto, se puede utilizar **URLConnection** para inspeccionar las propiedades del objeto remoto antes de transportarlo realmente al entorno local. Estos atributos son expuestos por la especificación del protocolo HTTP y, como tal, sólo tiene sentido para objetos **URL** que estén usando el protocolo HTTP.

URLConnection define varios métodos. A continuación se muestran algunos:

int getContentLength( )	Devuelve el tamaño en bytes de el contenido asociado con el recurso. Si la longitud no está disponible se devuelve –1.
String getContentType( )	Devuelve el tipo de contenido encontrado en el recurso. Este es el valor del **tipo de contenido** en el encabezado. Devuelve null si el tipo de contenido no está disponible.
long getDate( )	Devuelve el tiempo y datos de la respuesta representado en términos de milisegundos desde el primero de enero de 1970 tiempo GMT.
long getExpiration( )	Devuelve el día y la hora de expiración del recurso representado en términos de milisegundos desde el primero de enero de 1970 tiempo GMT. Se devuelve cero si los datos de expiración no están disponibles.

String getHeaderField(int *idx*)	Devuelve el valor del encabezado cuyo índice está dado por *idx*. Los índices de los encabezados comienza en 0. Devuelve null si el valor del índice excede el número de campos.
String getHeaderFiled(String *nombreCampo*)	Devuelve el valor del campo encabezado cuyo nombre está especificado por *nombreCampo*. Devuelve null si el nombre especificado no se encuentra.
String getHeaderFieldKey(int *idx*)	Devuelve la llave del encabezado del campo clave en el índice especificado por *idx*. Los encabezados de los índices comienzan en 0). Devuelve null si el valor de *idx* excede el número de campos.
Map<String, List<String>> getHeaderFields( )	Devuelve un mapa que contiene todos los campos de encabezado y sus valores.
Long getLastModified( )	Devuelve la fecha y hora, en milisegundos desde el primero de enero de 1970 tiempo GMT de la última modificación del recurso. Se devuelve cero si la fecha de última modificación no está disponible.
InputStream getInputStream( ) throws IOException	Devuelve un **InputStream** que está ligado a un recurso. Este flujo puede ser utilizado para obtener el contenido del recurso.

Note que **URLConnection** define varios métodos que gestionan la información de encabezados. Un encabezado consiste de pares de claves y valores representados como cadenas. Usando **getHeaderField( )**, se puede obtener el valor asociado con una clave de encabezado. Llamando **getHeaderFields( )**, se puede obtener un mapa que contiene todos los encabezados. Muchos encabezados estándar están disponibles directamente a través de métodos tales como **getDate( )** y **getContentType( )**.

En el siguiente ejemplo se crea una **URLConnection** utilizando el método **openConnection( )** de un objeto **URL**, y luego lo utiliza para examinar las propiedades y el contenido del documento:

```
// Ejemplo de URLConnection.
import java.net.*;
import java.io.*;
import java.util.Date;

class UCDemo
{
  public static void main(String args[]) throws Exception {
    int c;
    URL hp = new URL ("http://www.internic.net");
    URLConnection hpCon = hp.openConnection();

    // Obtiene fecha
    long d = hpCon.getDate();
    if (d == 0)
      System.out.println ("No hay información de la fecha");
    else
      System.out.println ("Fecha: " + new Date(d));
```

```
    // Obtiene tipo de contenido
    System.out.println ("Tipo del contenido: " + hpCon.getContentType());

    // Obtiene fecha de expiración
    d = hpCon.getExpiration();
    if (d == 0)
      System.out.println ("No hay información de expiración disponible");
    else
      System.out.println ("Expira: " + new Date(d));

    //Obtiene la fecha de última modificación
    d = hpCon.getLastModified();
    if (d == 0)
      System.out.println ("No hay información de última modificación disponible");
    else
      System.out.println ("Última modificación: " + new Date(d));

    //Obtiene el tamaño del contenido
    int len = hpCon.getContentLength();
    if (len == -1)
      System.out.println("Longitud del contenido no disponible");
    else
      System.out.println("Longitud del contenido: " + len);

    if (len != 0) {
      System.out.println("= = = Contenido = = =");
      InputStream input = hpCon.getInputStream();
      int i = len;
      while (((c = input.read()) != -1)) { //&& (--i > 0)) {
        System.out.print ((char) c);
      }
      input.close ();

    } else {
      System.out.println ("No hay contenido disponible");
    }

  }
}
```

El programa establece una conexión HTTP con **www.internic.net** a través del puerto 80. Luego saca los valores del encabezado y se obtiene el contenido. Aquí están las primeras líneas de la salida (la salida exacta variará dependiendo del tiempo).

```
Fecha: Thu Jan 17 19:43:58 CST 2008
Tipo del contenido: text/html
No hay información de expiración disponible
Última modificación: Mie Oct 05 19:49:29 CDT2005
Longitud del contenido: 4917
= = = Contenido = = =
<html>
<head>
<title>InterNIC | The Internet's Network Information Center</title>
<meta content="text/html; charset=utf-8" http-equiv=Content-Type>
<meta name="keywords"
       content="internic,network information, domain registration">
<style type="text/css">
```

```
<!--
p, li, td, ul { font-family: Arial, Helvetica, sans-serif}
-->
</style>
</head>
```

HttpURLConnection

Java provee una subclase de **URLConnection** que provee soporte para conexiones http. Esta clase es **HttpURLConnection**. Se puede obtener un **HttpURLConnection** en la misma forma que se acaba de mostrar, llamando **openConnection( )** sobre un objeto **URL**, pero debe convertir el tipo del resultado a **HttpURLCoonection** (claro que se nos debemos asegurar antes de estar abriendo una conexión HTTP). Una vez que se tiene la referencia al objeto **HttpURLConnection**, se pueden utilizar cualquiera de los métodos heredados de **URLConnection**. Y también se pueden utilizar alguno de los métodos definidos por **HttpURLConnection**. A continuación algunos ejemplos:

static boolean getFollowRedirects( )	Devuelve **verdadero** si el redireccionamiento se sigue automáticamente y **falso** en caso contrario. Esta característica está activada por omisión.
String getRequestMethod( )	Devuelve una cadena que representa cómo las solicitudes URL son realizadas. Por omisión es GET. Otras opciones, tales como POST están disponibles.
int getResponseCode( )throws IOException	Devuelve el código de respuesta de HTTP, se devuelve –1 si no es posible obtener el código de respuesta. Una excepción **IOException** se lanza si la conexión falla.
String getResponseMessage ( )throws IOException	Devuelve el mensaje de respuesta asociado con el código de respuesta. Devuelve null si no hay mensaje disponible. Una excepción **IOException** se genera cuando la conexión falla.
static void setFollowRedirects(boo lean *como*)	Si *como* es **verdadero**, entonces los redireccionamientos se siguen automáticamente. Si *como* es **falso**, los redireccionamientos no se siguen automáticamente. Los redireccionamientos se siguen por omisión.
void setRequestMethod(String *como*) throws ProtocolException	Establece el método mediante el cuál HTTP realiza las solicitudes. El método por omisión es GET, pero otras opciones, tales como POST, están disponibles. Si el valor de *como* es inválido, una excepción **ProtocolException** se genera.

El siguiente programa muestra el uso de **HttpURLConnection**. Primero establece una conexión a **www.google.com**. Luego despliega el método solicitante, el código de respuesta y el mensaje de respuesta. Finalmente despliega la clave y valores de respuesta del encabezado.

```
// Ejemplo de HttpURLConnection.
import java.net.*;
import java.io.*;
import java.util.*;
```

```
class HttpURLDemo
{
public static void main(String args[]) throws Exception {
    URL hp = new URL("http://www.google.com");

    HttpURLConnection hpCon = (HttpURLConnection) hp.openConnection();

    // Muestra el método solicitante
    System.out.println("El método solicitante es: " +
                       hpCon.getRequestMethod());

    // Muestra el código de respuesta
    System.out.println("El código de respuesta es: " +
                       hpCon.getResponseCode()) ;

    // Muestra el mensaje de respuesta.
    System.out.println("El mensaje de respuesta es: "+
                       hpCon.getResponseMessage()) ;

    // Obtiene una lista de encabezados y
    // fija los encabezados clave
    Map<String, List<String>> hdrMap = hpCon.getHeaderFields();
    Set<String> hdrField = hdrMap.keySet();

    System.out.println("\n Aquí está el encabezado:");

    // Muestra todos los encabezados clave y sus valores.
    for(String k : hdrField) {
      System.out.println ("Clave: " + k +
                          " Valor: " + hdrMap.get (k));
    }
  }
}
```

La salida producida por el programa se muestra a continuación. Claro que, la respuesta exacta devuelta por **www.google.com** puede variar.

```
El método solicitante es GET
El código de respuesta es 200
El mensaje de respuesta es OK

Aquí está el encabezado:
Clave: Set-Cookie Valor:
[PREF=ID=4fbe93944led966b:TM=1150213711:LM=1150213711:S=Qk81
WCVtvYkJOdh3; expires=Sun, 17-Jan-2038 19:14:07 GMT; path=/;
  domain=.google.com]
Clave: null Valor: [HTTP/1.1 200 OK]
Clave Date Valor: [Tue, 13 Jun 2006 15:48:31 GMT]
Clave: Content-Type Valor: [text/html]
Clave: Server Valor: [GWS/2.1]
Clave: Transfer-Encoding Valor: [chunked]
Clave: Cache-Control Valor: [private]
```

Nótese cómo las claves y valores del encabezado se muestran en pantalla. Primero, se obtienen las claves y valores de los encabezados en una estructura de datos map llamando el método **getHeaderFIelds( )** (el cual es heredado de **URLConnection**). A continuación, un conjunto de claves de encabezado es recuperado llamando a **keySet( )** sobre la estructura map.

Luego el conjunto de claves se recorre utilizando un ciclo estilo **for-each**. El valor asociado con cada clave se obtiene llamando al método **get( )** de map.

La Clase URI

Una adición relativamente reciente a Java es la clase **URI**, la cual encapsula un *Identificador de Recursos Uniforme* (URI por sus siglas en inglés). Los URI son similares a los URL. De hecho los URL constituyen un subconjunto de los URI. Un URI representa una forma estándar de identificar un recurso. Un URL también describe cómo acceder al recurso.

Cookies

El paquete **java.net** incluye clases e interfaces que ayudan a administrar cookies y puede ser utilizado para crear una sesión HTTP. Las clases son **CookieHandler**, **CookieManager** y **HttpCookie**. Las interfaces son **CookiePolicy** y **CookieStore**. Todos, excepto **CookieHandler** fueron agregadas por Java SE 6. (**CookieHandler** fue agregado por JDK5). La creación de una sesión HTTP está más allá del alcance de este libro.

NOTA. *Para información del uso de cookies con servlets, véase el Capítulo 31.*

Conectores TCP/IP para servidores

Como hemos mencionado antes, Java tiene una clase de socket diferente que se debe utilizar para crear aplicaciones de servidor. La clase **ServerSocket** se utiliza para crear servidores que escuchen a programas locales o remotos que se conecten con ellos en puertos publicados. Los **ServerSockets** son un poco diferentes de los **Sockets** normales. Cuando se crea un **ServerSocket**, se registrará por sí mismo con el sistema para recibir conexiones de sockets clientes. El constructor para el **ServerSocket** refleja el número de puerto del que se desea aceptar conexiones y opcionalmente por cuanto tiempo quiere que la fila de dicho puerto esté disponible. El tamaño de la fila le dice al sistema cuantas conexiones de clientes puede dejar pendientes antes de simplemente declinar las conexiones. Por omisión son 50 conexiones. Los constructores podrían enviar una **IOException** bajo condiciones adversas. Aquí se presentan tres de los constructores mencionados.

ServerSocket(int *puerto*) throws IOException	Crea un socket servidor en el puerto especificado con longitud de fila igual a 50.
ServerSocket(int *puerto*, int *maxFila*) throws IOException	Crea un socket servidor en el puerto especificado con longitud de fila máxima *maxFila*.
ServerSocket(int *puerto*, int *maxFila*, InetAddress *direccionLocal*) throws IOException	Crear un socket servidor en el puerto especificado con longitud de fila máxima *maxFila*. En un nodo con múltiples direcciones, *direccionLocal* especifica la dirección IP al que se liga a este socket.

ServerSocket tiene un método adicional llamado **accept( )**, que es una llamada que se bloquea y esperará a que un cliente inicie comunicaciones, y después devuelve un **Socket** normal que se utilizará para la comunicación con el cliente.

Datagramas

El estilo de trabajo en red TCP/IP es apropiado para la mayoría de las necesidades. Proporciona un flujo de paquetes serializado, predecible y fiable. No obstante, esto tiene su costo. TCP incluye muchos algoritmos complicados para manejar el control de congestión en redes muy concurridas, así como expectativas pesimistas sobre pérdida de paquetes. Esto da lugar un modo algo ineficiente de transportar datos. Los datagramas ofrecen una alternativa.

Los *datagramas* son bloques de información pasados entre máquinas. Son algo así como un lanzamiento rápido de un catcher de béisbol bien entrenado, pero con los ojos vendados, a la tercera base. Una vez que el datagrama ha sido emitido hacia su objetivo pretendido, no hay seguridad de que llegará, o incluso de que habrá alguien ahí para recogerlo. Análogamente, cuando el datagrama es recibido, no hay seguridad de que no se ha dañado durante el trayecto o que quienquiera que lo enviara sigue ahí para recibir una respuesta.

Java implementa los datagramas sobre el protocolo UDP utilizando dos clases: el objeto **DatagramPacket** es el contenedor de datos, mientras que el **DatagramSocket** es el mecanismo utilizado para enviar o recibir los paquetes **DatagramPacket**. A continuación se examina cada uno.

DatagramSocket

DatagramSocket define cuatro constructores públicos. Los cuales se muestran a continuación:

DatagramSocket( ) throws SocketException

DatagramSocket(int *puerto*) throws SocketException

DatagramSocket(int *puerto*, InetAddress *ipDireccion*) throws SocketException

DatagramSocket(SocketAddresss *direccion*) throws SocketException

El primero crea un **DatagramScoket** que se conecta a cualquier puerto sin uso sobre la computadora local. El segundo crea un **DatagramScoket** que se conecta al puerto especificado en el parámetro puerto. El tercero construye un **DatagramScoket** que se conecta al puerto y la **InetAddres** especificados. El cuarto construye un **DatagramScoket** que se conecta al **SocketAddres** especificado. **SocketAddress** es una clase abstracta que se implementa por la clase concreta **InetSocketAddress**. **InetSocketAddress** encapsula una dirección IP y un número de puerto. Todos pueden lanzar una **SocketException** si ocurre algún error mientras se crea el socket.

DatagramSocket define varios métodos. Dos de los más importantes son **send( )** y **receive( )**, los cuales se muestran a continuación:

void send(DatagramaPacket *paquete*) throws IOException

void receive(DatagramaPacket *paquete*) throws IOException

PARTE II

El método **send( )** envía un paquete de datos al puerto especificado por el parámetro *paquete*. El método **receive( )** espera para recibir un paquete del puerto especificado por el parámetro *paquete* y devuelve un resultado.

Otros métodos proveen acceso a varios atributos asociados con un **DatagramSocket**. Aquí están algunos de ellos

InetAddress getInetAddress( )	Si el socket está conectado, entonces se devuelve la dirección, en otro caso, se devuelve null.
int getLocalPort( )	Devuelve el número del puerto local.
int getPort( )	Devuelve el número del puerto al cuál el socket está conectado. Devuelve -1 si el socket no está conectado a un puerto.
boolean isBound( )	Devuelve **verdadero** si el socket está conectado a una dirección. Devuelve **falso** en cualquier otro caso.
boolean isConnected( )	Devuelve **verdadero** si el socket está conectado a un servidor. Devuelve **falso** en cualquier otro caso.
void setSoTimeout(int *m*) throws SocketException	Fija el periodo de tiempo de espera en *m milisegundos*.

DatagramPacket

DatagramPacket define varios constructores. Cuatro de ellos se muestran a continuación:

DatagramPacket(byte *datos*[ ], int *tamaño*)
DatagramPacket(byte *datos*[J, int *pos*, int *tamaño*)
DatagramPacket(byte *datos*[ ], int *tamaño*, InetAddress *direccionlP*, int *puerto*)
DatagramPacket(byte *datos*[ ], int *pos*, int *tamaño*, InetAddress *direccionlP*, int *puerto*)

El primer constructor especifica un buffer que recibirá datos, y el tamaño de un paquete. Se utiliza para recibir datos por un socket **DatagramSocket**. La segunda forma permite especificar una posición dentro del buffer en que se almacenarán los datos. La tercera forma especifica una dirección y un puerto objetivo, que son utilizaos por un **DatagramSocket** para determinar a dónde se enviarán los datos del paquete. La cuarta forma transmite paquetes comenzando en la posición especificada dentro del arreglo datos. Piense en las dos primeras formas como en construir un"buzón de entrada", y en las dos últimas como en llenar un sobre y ponerle la dirección.

DatagramPacket define varios métodos, incluyendo los que se muestran aquí, que dan acceso a la dirección de destino y al número de puerto de un paquete, así como a los datos brutos y su longitud. En general, el método **get** es utilizado sobre paquetes que se están recibiendo y el métodos **set** es utilizado sobre paquetes que serán enviados.

InetAddress getAddress( )	Devuelve la dirección de la fuente (para los datagramas que se están recibiendo) o destino (para los datagramas que se están enviando).
byte[ ] getData( )	Devuelve el arreglo de datos de tipo byte contenidos en el datagrama. En su mayoría es utilizado para recuperar los datos de los datagramas después de que han sido recibidos.

int getLength( )	Devuelve el tamaño de los datos válidos contenidos en el arreglo de bytes que serían devueltos por el método **getData( )**. Esto podría no ser igual al tamaño del arreglo completo.
int getOffset( )	Devuelve el índice de inicio de los datos.
int getPort( )	Devuelve el número del puerto.
void setAddress(InetAddress *direccionIP*)	Fija la dirección a la cuál un paquete será enviado. La dirección está especificada por *direccionIP*.
void setData(byte[ ] *datos*)	Coloca datos en el campo datos, la posición de inicio de índice en cero y el tamaño al número de bytes en *datos*.
void setData(byte[ ] *data*, int *idx*, int *tamaño*)	Coloca datos en el campo datos, la posición de inicio de índice en *idx* y el tamaño al número de bytes en *tamaño*.
void setLength(int *tamaño*)	Establece el tamaño de un paquete a través del argumento *tamaño*.
void setPort(int *puerto*)	Establece el puerto con el valor dado en el argumento *puerto*.

Un ejemplo utilizando Datagramas

El siguiente ejemplo implementa un cliente y un servidor de comunicaciones en red muy simples. Los mensajes se teclean en la ventana del servidor y se escriben a través de la red del lado del cliente, donde se imprimen.

```
// Ejemplo de los datagramas.
import java.net.*;

class WriteServer {
  public static int serverPort = 998;
  public static int clientPort = 999;
  public static int buffer_size = 1024;
  public static DatagramSocket ds;
  public static byte buffer[] = new byte[buffer_size];

  public static void TheServer() throws Exception {
    int pos=0;
    while (true) {
      int c = System.in.read();
      switch (c) {
        case -1:
          System.out.println("El servidor termina.");
          return;
        case '\r':
          break;
        case '\n':
          ds.send(new.DatagramPacket(buffer, pos,
            InetAddress.getLocalHost(),clientPort));
          pos=0;
          break;
        default:
          buffer [pos++] = (byte) c;
    }
  }
}
  public static void TheClient() throws Exception {
```

```
      while (true) {
      DatagramPacket p = new DatagramPacket(buffer, buffer.length);
      ds.receive(p);
      System.out.println(new String(p.getData(), 0, p.getLength()));
    }
  }

  public static void main(String args[]) throws Exception {
    if (args.length = = 1) {
      ds = new DatagramSocket(serverPort);
      TheServer();
    } else {
      ds = new DatagramSocket(clientPort);
      TheClient() ;
    }
  }
}
```

Este programa ejemplo está limitado por el constructor de **DatagramSocket** a ejecutarse entre dos puertos de la máquina local. Para utilizar el programa se ejecuta

```
java WriteServer
```

en una ventana; este será el cliente. Y luego se ejecuta

```
java WriteServer 1
```

Éste será el servidor. Cualquier cosa tecleada en la ventana del servidor será enviada a la ventana del cliente tras recibirse un carácter de nueva línea.

CAPÍTULO 21

La clase Applet

Este capítulo examina la clase **Applet**, que proporciona lo necesario para programar applets. La clase **Applet** está contenida en el paquete **java.applet** y tiene varios métodos que proporcionan un control completo de la ejecución de los applets. Además, **java.applet** define tres interfaces: **AppletContext**, **AudioClip** y **AppletStub**.

Dos tipos de applets

Es importante comenzar mencionando que existen dos variantes de applets. La primera variante se conforma por los applets basados directamente en la clase **Applet**, los cuales se describen en este capítulo. Estos applets utilizan el AWT para proporcionar la interfaz gráfica de usuario. Este estilo de applets ha estado disponible desde la creación de Java.

El segundo tipo de applets, es el que se basa en la clase **JApplet** de Swing. Los applets de Swing utilizan las clases de Swing para crear la interfaz gráfica. Swing proporciona interfaces de usuario más ricas y fáciles de utilizar que AWT. Por ello, los applets basados en Swing son los más populares. Sin embargo los applets tradicionales basados en AWT son utilizados aún, en particular cuando sólo se necesita una interfaz de usuario simple. Por ello, ambos applets los basados en AWT y los basados en Swing se utilizan actualmente.

Debido a que **JApplet** hereda de **Applet**, todas las características de **Applet** están disponibles también en **JApplet**, y mucha de la información de este capítulo aplica para ambos tipos de applets. Por consiguiente, aunque solo nos interesen los applets de Swing, la información de este capítulo es relevante y necesaria. Sin embargo debemos considerar que cuando se construyan applets basados en Swing será necesario considerar algunas restricciones adicionales las cuales se describen más adelante en este libro, en el capítulo donde se habla de Swing.

NOTA *Para saber más sobre la construcción de applets basados en Swing consúltese el Capítulo 29.*

Fundamentos de Applet

El Capítulo 13 presentó la forma general de un applet y los pasos necesarios para compilar y ejecutar uno. Comencemos revisando esa información.

Todos los applets son subclases de la clase **Applet** (directa o indirectamente). Los applets no son programas independientes. Los applets requieren para su ejecución un navegador Web o un visor de applets. Las figuras que se pueden ver en este capítulo se han creado con el visor de applets estándar,

llamado **appletviewer**, incluido en el JDK. Pero es posible utilizar cualquier otro visor de applets o navegador.

La ejecución de un applet no comienza en **main( ).** Realmente, pocos son los applets que tienen método **main( )**. El mecanismo que arranca y controla la ejecución de un applet es totalmente diferente y se explicará más adelante.

La salida a la ventana de applets no se realiza por **System.out.println( )**, sino que es gestionada por varios métodos del AWT, tal como **drawString( )**, que presenta en pantalla una cadena en la coordenada X,Y especificada. La entrada también se gestiona de forma diferente que en las aplicaciones de consola (Recordemos que en los applets basados en Swing se utilizan las clases Swing para manejar las interacciones con el usuario, esto se describe más adelante en este libro).

Una vez que se ha compilado un applet, se incluye en un archivo HTML utilizando la etiqueta APPLET (también puede emplearse la etiqueta OBJECT pero Sun recomienda el uso de la etiqueta APPLET y es la etiqueta que utilizaremos en este libro) El applet se ejecutará en un navegador compatible con Java cuando éste encuentre la etiqueta APPLET en el archivo HTML. Para ver y probar más fácilmente un applet, simplemente hay que incluir un comentario en el encabezado del archivo de código fuente Java que contenga la etiqueta APPLET. De esta forma, el código tiene las sentencias HTML que necesitará el applet, y se podrá probar el applet compilado arrancando el visor de applets con el archivo de código fuente de Java. A continuación, un ejemplo de lo comentado:

```
/*
<applet code="MiApplet" width=200 height=60>
</applet>
*/
```

Este comentario contiene una etiqueta APPLET que ejecutará un applet llamado **MiApplet** en una ventana de 200 píxeles de ancho por 60 píxeles de alto. Todas los applets mostrados en este libro contienen una etiqueta APPLET incluida en un comentario. Trabajar de esta manera facilita probar los applets.

La clase Applet

La clase **Applet** define los métodos mostrados en la Tabla 21-1. **Applet** proporciona lo necesario para la ejecución de un applet, desde el arranque hasta la finalización. También proporciona métodos que cargan y visualizan imágenes y métodos que cargan y permiten oír archivos de audio. **Applet** extiende la clase **Panel**. Además, **Panel** hereda la clase **Container**, y ésta hereda de la clase **Component**. Estas clases proporcionan la base para desarrollar interfaces gráficas basadas en ventanas en Java. Así, **Applet** proporciona todo el soporte necesario para las actividades basadas en ventanas. AWT será descrito detalladamente en los siguientes capítulos.

Método	Descripción
void destroy( )	Método llamado por el navegador justo antes de que termine el applet. El applet debe sobrescribir este método si necesita liberar algún recurso antes de finalizar.
AccessibleContext getAccessibleContext( )	Devuelve el contexto de accesibilidad del objeto.

TABLA 21-1 Métodos definidos por **Applet**

Método	Descripción
AppletContext getAppletContext( )	Devuelve el contexto asociado al applet.
String getAppletInfo( )	Devuelve una cadena que describe al applet
AudioClip getAudioClip(URL *url*)	Devuelve un objeto **AudioClip** que encapsula el archivo de audio encontrado en la dirección especificada por *url*.
AudioClip getAudioClip(URL *url*, String *clipName*)	Devuelve un objeto **AudioClip** que encapsula el archivo de audio encontrado en la dirección especificada por *url* y que tiene el nombre especificado por *clipName*.
URL getCodeBase( )	Devuelve el URL asociado con el applet que llama al método.
URL getDocumentBase( )	Devuelve el URL del documento HTML que invoca al applet.
Image getImage(URL *url*)	Devuelve un objeto **Image** que encapsula la imagen encontrada en la dirección especificada por *url*.
Image getImage(URL *url*, String *imageName*)	Devuelve un objeto **Image** que encapsula la imagen encontrada en la dirección especificada por *url* y que tiene el nombre especificado por *imageName*.
Locale getLocale( )	Devuelve un objeto **Locale** que se utiliza por varias clases y métodos que pueden trabajar con ese tipo de objetos.
String getParameter(String *paramName*)	Devuelve el parámetro asociado con *paramName*. Devuelve **null** si no se encuentra el parámetro especificado.
String[ ] [ ] getParameterInfo( )	Devuelve una tabla de valores tipo **String** que describe los parámetros reconocidos por el applet. Cada entrada de la tabla consta de tres cadenas que contienen el nombre del parámetro, una descripción de su tipo y/o rango, y una explicación de su propósito respectivamente.
void init( )	Este método es llamado cuando un applet comienza a ejecutarse. Es el primer método que se ejecuta en un applet.
boolean isActive( )	Devuelve **true** si ha comenzado el applet. Devuelve **false** si se ha parado o detenido el applet.
static final AudioClip newAudioClip (URL *url*)	Devuelve un objeto **AudioClip** que encapsula el archivo de audio encontrado en la dirección especificada por *url*. Este método es similar a **getAudioClip()** excepto en que éste es estático y se puede ejecutar sin la necesidad de un objeto **Applet**.
void play(URL *url*)	Si se encuentra un archivo de sonido en la dirección especificada por *url*, lo reproduce.
void play(URL *url*, String *clipName*)	Si se encuentra un archivo de sonido en la dirección *url* con el nombre especificado por *clipName*, lo reproduce.
void resize(Dimension *dim*)	Cambia el tamaño del applet en función de las dimensiones especificadas por *dim*. **Dimension** es una clase de **java.awt** que contiene dos campos enteros: **width** y **height**.
void resize(int *width*, int *height*)	Cambia el tamaño del applet en función de las dimensiones especificadas por *width* y *height*.

TABLA 21-1 Métodos definidos por **Applet** (*continuación*)

Método	Descripción
final void setStub(AppletStub *stubObj*)	Hace que *stubObj* sea el stub del applet. Este método lo utiliza el intérprete de Java y normalmente no lo utiliza el applet. Un *stub* es una pequeña parte de código que proporciona el enlace entre el applet y el navegador.
void showStatus(String *str*)	Muestra la cadena *str* en la ventana de estado del navegador o del visor de applets. Si el navegador no dispone de una ventana de estado, entonces no se realiza ninguna acción.
void start( )	El navegador llama a este método cuando debe comenzar (o continuar) la ejecución de un applet. Cuando los applets comienzan por primera vez, se ejecuta automáticamente después de **init( )**
void stop( )	El navegador llama a este método para parar o detener la ejecución del applet. Una vez parado, un applet puede volver a ejecutarse cuando el navegador llama al método **start( )**

TABLA 21-1 Métodos definidos por **Applet** (*continuación*)

Arquitectura de un Applet

Un applet es un programa basado en ventanas. Por ello, su arquitectura es diferente de la de los programas basados en consola mostrados en la primera parte de este libro. Si está familiarizado con la programación bajo Windows, no tendrá problemas en escribir applets. En caso contrario hay algunos conceptos clave que debemos comprender.

En primer lugar, los applets son conducidos u orientados a eventos. Aunque no se verá la gestión de eventos sino hasta el siguiente capítulo, es importante tener una idea aproximada de cómo ésta modifica la manera de diseñar un applet. Un applet se parece a un conjunto de rutinas de tratamiento de interrupciones. A continuación, se explica cómo funciona el proceso. El applet espera hasta que ocurre un evento. El intérprete notifica del evento al applet llamando a un gestor de eventos para que lo trate. Después, el applet debe ejecutar la acción que desee el programador y devolver el control rápidamente. Éste es un punto muy importante. Para la gran mayoría de los casos, el applet no debería entrar en un nuevo "modo" de operación que mantenga el control durante un largo periodo de tiempo. Todo lo contrario, el applet debe realizar acciones específicas en respuesta a eventos, y después devolver el control al intérprete. En las situaciones en las que el applet necesite realizar una tarea repetitiva (por ejemplo, mostrando un mensaje de varias líneas en su ventana), se debería iniciar un nuevo hilo que ejecute esa tarea. Un ejemplo de ello se verá más adelante en este mismo capítulo.

En segundo lugar, es el usuario el que inicia una interacción con un applet, y no al revés. Como ya es sabido, en un programa sin ventanas, cuando el programa necesita una entrada la solicita al usuario y se llama a algún método de entrada, tal como **readLine( )**. Un applet no trabaja así, sino que es el usuario el que interactúa con el applet cómo y cuándo él quiera. Estas interacciones se mandan al applet como eventos a los que el applet debe responder. Por ejemplo, cuando el usuario hace clic dentro de la ventana de un applet, se genera un evento clic. Si el usuario presiona una tecla cuando la ventana del applet está activa, se genera un evento keyPress. Como se verá en posteriores capítulos, los applets pueden contener varios controles, como botones o cajas de texto. Cuando el usuario interactúa con uno de esos controles, se genera un evento.

Ya que la arquitectura de un applet no es tan fácil de entender como la de un programa basado en consola, Java la hace lo más simple posible. Si se han escrito programas para Windows, sabrá lo complejo que puede llegar a ser ese entorno. Afortunadamente, Java proporciona un acceso mucho más claro y sencillo.

Estructura de un Applet

Casi todos los applets pequeños sobrescriben una serie de métodos que proporcionan el mecanismo básico mediante el que el navegador o el visor de applets se conectan con el applet y controla su ejecución. Cuatro de estos métodos, **init( )**, **start( )**, **stop( )** y **destroy( )**, están definidos por **Applet**. Para todos estos métodos se proporcionan implementaciones por omisión. Los applets no necesitan sobrescribir los métodos que no utilizan. Sin embargo, sólo los applets muy sencillos no necesitan definir todos estos métodos.

Los applets basados en AWT (como los mostrados en este capítulo) también sobrescribirán el método **paint( )**, definido en la clase **Component** de AWT. Este método se llama cuando la salida del applet deba ser desplegada nuevamente. Los applets basados en Swing utilizan un mecanismo diferente para realizar esta labor. Estos cinco métodos se ensamblan como se ve en la siguiente estructura:

```
// Estructura de un applet.
import java.awt.*;
import java.applet.*;
/*
<appletcode="EsqueletoDeApplet" width=300 height=100>
</applet>
*/

public class EsqueletoDeApplet extends Applet {
  // Este es el método que se llama en primer lugar.
  public void init() {
    // código de inicialización
  }

  /* Este método es llamado en segundo lugar después de init().
     También es llamado siempre que se reanuda el applet. */
  public void start() {
    // código que comienza o reanuda la ejecución
  }

  // Este método se ejecuta cuando se detiene el applet.
  public void stop() {
    // código que detiene la ejecución
  }

  /* Este método se ejecuta cuando termina el applet.
     Es el último método que se ejecuta*/
  public void destroy() {
    // código para la finalización definitiva
  }

  // Este método es llamado cuando se restaura la ventana del applet.
  public void paint(Graphics g) {
     // código para dibujar el contenido de la ventana
  }
}
```

PARTE II

Aunque este programa no hace nada, se puede compilar y ejecutar. Cuando se ejecuta, genera la siguiente ventana vista con un visor de applets:

Comienzo y final de un applet

Es importante entender el orden en el que se llama a los diferentes métodos mostrados en la estructura del applet. Cuando comienza un applet, el AWT llama a los siguientes métodos en el siguiente orden:

1 **init( )**

2 **start( )**

3 **paint( )**

Cuando finaliza un applet, se llama a los siguientes métodos en este orden:

1 **stop( )**

2 **destroy( )**

A continuación se verán más detalladamente estos métodos.

init( )

El método **init( )** es el primer método al que se llama. Es aquí donde se deberían inicializar las variables. A este método sólo se le llama una única vez durante la ejecución del applet.

start( )

Al método **start( )** se le llama inmediatamente después del método **init( )**. También es llamado para reanudar un applet después de que éste se haya detenido. Mientras que a **init( )** sólo se le llama una vez (cuando inicialmente se carga un applet) a **start( )** se le llama cada vez que un documento HTML de un applet se visualiza en la pantalla. Por lo tanto, si un usuario deja una página Web y vuelve atrás, el applet continúa la ejecución en **start( )**.

paint( )

Al método **paint( )** se le llama cada vez que la salida del applet tiene que redibujarse. Esto puede ocurrir por diversas razones. Por ejemplo, si otra ventana tapa la ventana en la que está corriendo el applet y hay que visualizarla de nuevo. O cuando se minimiza la ventana del applet y después se restaura. A **paint( )** también se le llama cuando el applet comienza a ejecutarse. Cualquiera que sea la causa, siempre que el applet tenga que volver a dibujar su salida, se llama a **paint( )**. El método **paint( )** tiene un parámetro del tipo **Graphics**. En este parámetro estará el contexto gráfico, que describe el entorno gráfico en el que se está ejecutando el applet. Este contexto se usa siempre que se requiera una salida al applet.

stop( )

Al método **stop( )** se le llama cuando un navegador deja el documento HTML que contiene al applet , por ejemplo cuando se va a otra página. Cuando se llama a **stop( )**, probablemente el applet continúa ejecutándose. Se debería usar **stop( )** para suspender tareas que no necesiten ejecutarse cuando el applet no es visible. Se pueden reanudar esas tareas cuando se llama a **start( )**, esto es, si el usuario vuelve a la página.

destroy( )

Al método **destroy( )** se le llama cuando el entorno determina que el applet necesita ser borrado completamente de la memoria. Aquí es cuando hay que liberar cualquier recurso que pueda estar utilizando el applet. Al método **stop( )** se le llama siempre antes que a **destroy( )**.

Sobrescribir el método update( )

En algunas situaciones un applet puede necesitar sobrescribir otro método definido por AWT llamado **update( )**. A este método se le llama cuando el applet ha pedido que se redibuje parte de su ventana. La versión por omisión de **update( )** simplemente llama a **paint( )**. Sin embargo se puede sobrescribir el método **update( )** para que realice un repintado más sutil. En general, sobrescribir al método **update( )** es una técnica especializada que no es apropiada para todos los applets. Los ejemplos en este libro no realizan sobreescritura de **update( )**.

Métodos sencillos de visualización de applets

Como se ha mencionado anteriormente, los applets se visualizan en una ventana y los applets basados en AWT utilizan métodos de AWT para realizar las entradas y salidas. Aunque los métodos, procedimientos y técnicas necesarias para una completa gestión del entorno de ventanas con AWT se verán en capítulos posteriores, se van a describir aquí unos pocos, ya que los usaremos para escribir algunos ejemplos de applets. Recuerde que los applets basados en Swing se describen más adelante en este libro.

Como se describió en el Capítulo 13, para mostrar una cadena en la ventana de un applet, se utiliza **drawString( )**, que es un miembro de la clase **Graphics**. Normalmente, se le llama desde **update( )** o desde **paint( )**. Su forma general es la siguiente:

void drawString(String *message*, int *x*, int *y*)

Aquí, *message* es la cadena a mostrar en la posición dada por *x*, *y*. En una ventana en Java, la esquina superior izquierda es el punto 0,0. El método **drawString( )** no reconoce caracteres de salto de línea. Si se quiere comenzar una nueva línea de texto, hay que hacerlo manualmente, especificando la coordenada X,Y exacta donde se quiere que comience la línea. Como se verá en posteriores capítulos, existen técnicas para realizar este proceso fácilmente.

Para establecer el color de fondo de la ventana de un applet, se utiliza **setBackground( )**. Para establecer el color del frente (por ejemplo, el color para el texto), se utiliza **setForeground( )**. Estos métodos se definen en la clase **Component**, y sus formas generales son las siguientes:

void setBackground(Color *newColor*)

void setForeground(Color *newColor*)

Aquí, *newColor* especifica el nuevo color. La clase **Color** define las constantes que se muestran a continuación y que pueden ser utilizadas para especificar colores:

Color.black	Color.magenta
Color.blue	Color.orange
Color.cyan	Color.pink
Color.darkGray	Color.red
Color.gray	Color.white
Color.green	Color.yellow
Color.lightGray	

Las constantes también están definidas con sus nombres completamente en mayúsculas.

El siguiente ejemplo pone el color de fondo verde y el del texto en rojo:

```
setBackground(Color.green);
setForeground(Color.red);
```

Un buen lugar para establecer el color de fondo y el del frente es en el método **init( ).** Por supuesto que estos colores se pueden cambiar tantas veces como se necesite durante la ejecución del applet.

Se pueden obtener los colores que actualmente están establecidos para el fondo y el frente llamando a los métodos **getBackground( )** y **getForeground( )**, respectivamente. También estos están definidos por la clase Component y se muestran a continuación:

Color getBackground( )
Color getForeground( )

Se muestra ahora un applet sencillo que establece el color de fondo en cyan, el color de frente en rojo, y visualiza un mensaje que ilustra el orden en que se llama a los métodos **init( )**, **start( )** y **paint( )** cuando comienza un applet:

```
/* Un applet que establece los colores de fondo y frente
   y muestra una cadena en la pantalla. */
import java.awt.*;
import java.applet.*;
/*
<applet code="Ejemplo" width=300 height=50>
</applet>
*/

public class Ejemplo extends Applet{
  String msg ;

  // Establece los colores de frente y fondo.
  public void init() {
    setBackground(Color.cyan);
    setForeground(Color.red);
    msg = "En el interior de init() --";
  }

  // Inicializa el texto que va a ser mostrado.
  public void start() {
    msg += "En el interior de start() --";
  }

    // Muestra msg en la ventana del applet.
    public void paint(Graphics g) {
```

```
    msg += " En el interior de paint(). ";
    g.drawString(msg, 10, 30);
  }
}
```

Este applet genera la siguiente ventana:

No se sobrescriben los métodos **stop( )** y **destroy( )**, ya que no son necesarios en este sencillo applet.

Repintar la pantalla

Como regla general, un applet escribe en su ventana sólo cuando el AWT llama a los métodos del applet **update( )** o **paint( )**. Esto suscita una pregunta interesante: ¿cómo puede causar un applet por sí mismo que su ventana se actualice cuando su información cambia? Por ejemplo, si en un applet se está mostrando una frase en movimiento, ¿cuál es el mecanismo que usa el applet para actualizar la ventana en cada momento en que esa frase se mueve? Hay que recordar una de las restricciones fundamentales impuestas en un applet, es que éste debe devolver rápidamente el control al intérprete. No se puede crear un ciclo dentro de **paint( )** que, por ejemplo, mueva repetidamente la frase. Esto podría impedir que se devolviera el control al AWT. Dada esta restricción, puede parecer que sacar algo en el applet es algo de lo más complicado. Afortunadamente, no es así. Siempre que el applet necesite actualizar la información que saca en su ventana, simplemente llama al método **repaint( )**.

El AWT es el que define al método **repaint( )**. Esto hace que el sistema ejecute una llamada al método **update( )** del applet, y este método, por omisión, llama a **paint( )**. Para que otra parte del applet presente algo a su ventana, simplemente hay que guardar esa salida y llamar a **repaint( )**. Entonces, el AWT ejecutará una llamada a **paint( )**, que visualiza la información guardada. Por ejemplo, si parte del applet necesita mostrar una cadena de texto en la pantalla, puede guardar la cadena en una variable **String** y después llamar a **repaint( )**. Dentro de **paint( )**, se puede mostrar la cadena utilizando **drawString( )**.

El método **repaint( )** tiene cuatro formas. Veamos por orden una a una. La versión más simple de **repaint( )** es la siguiente:

```
void repaint( )
```

Esta versión hace que toda la ventana sea repintada. La siguiente versión especifica una región que será repintada:

```
void repaint(int left, int top, int width, int height)
```

Aquí, las coordenadas de la esquina superior izquierda de la región se especifican por *left* y *top*, y el ancho y alto de la región se especifican por *width* y *height*. Estas dimensiones se especifican en píxeles. Se ahorra tiempo especificando una región a repintar. Las actualizaciones de toda

la ventana suponen un costo en tiempo. Si sólo se necesita actualizar una pequeña parte de la ventana, es más eficiente repintar sólo esa región.

Llamar a **repaint( )** es, esencialmente, una petición para que el applet sea repintado a la brevedad. Sin embargo, si el sistema es lento o está ocupado, podría no llamarse inmediatamente a **update( )**. Demasiadas peticiones de repintar en poco tiempo pueden colapsar el AWT; por esta razón, a **update( )** sólo se le debe llamar esporádicamente. Esto puede ser un problema en muchas situaciones, incluida la animación, donde la actualización continua es crucial. Una solución a este problema es utilizar las siguientes formas de **repaint( )**:

void repaint(long *maxRetraso*)
void repaint(long *maxRetraso*, int *x*, int *y*, int *ancho*, int *alto*)

maxRetraso especifica el número máximo de milisegundos que pueden transcurrir antes de llamar a **update( )**. Pero cuidado, si el tiempo que transcurre antes de que se haya llamado a **update( )** es mayor que maxRetraso, la llamada no se realizará. No se genera un valor de retorno ni se lanza ninguna excepción, por lo que se debe tener cuidado.

---

*NOTA Es posible mostrar algo en una ventana del applet con un método distinto a **paint( )** o **update( )**. Para hacerlo, hay que obtener el contexto gráfico llamando al método **getGraphics( )** (definido por **Component**) y usarlo para mostrar lo que se desee en la ventana. Sin embargo, en la mayoría de las aplicaciones, es mejor y más fácil mostrar algo a través de paint( ) y llamar a repaint() cuando cambie el contenido de la ventana.*

Un Applet sencillo

Para demostrar el uso de **repaint( )**, escribiremos un applet de ejemplo. Este applet muestra un mensaje que se mueve de derecha a izquierda a través de la ventana del applet.Ya que esto es una tarea repetitiva, se realiza en un hilo separado, creado por el propio applet cuando es inicializado. El applet es el siguiente:

```
/* Un applet sencillo.

   Este applet crea un hilo que desplaza
   el mensaje contenido en la variable msg de derecha a izquierda
   en la ventana del applet.
*/
import java.awt.*;
import java.applet.*;
/*
<applet code="AppletDesplazarFrase" width=300height=50>
</applet>
*/

  public class AppletDesplazarFrase extends Applet implements Runnable {
    String msg = " Una frase desplazándose.";
    Thread t = null;
    int state;
    boolean stopFlag;

    // Establece el color e inicializa el hilo.
    public void init() {
      setBackground(Color.cyan);
      setForeground(Color.red);
    }
```

```
    // Comienza el hilo
    public void start() {
      t = new Thread(this) ;
      stopFlag = false;
      t.start();
  }
  // Punto de entrada del hilo que desplaza la frase.
  public void run() {
    char ch;
    // Muestra la frase
    for( ; ; ) {
      try {
        repaint();
        Thread.sleep(250);
        ch = msg.charAt(0);
        msg = msg.substring(1, msg.length());
        msg += ch;
        if(stopFlag)
          break;
      } catch(InterruptedException e) {}
    }
  }
  // Detiene el desplazamiento de la frase.
  public void stop() {
    stopFlag = true;
    t = null;
  }
  // Muestra en pantalla la frase.
  public void paint(Graphics g) {
    g.draWString(msg, 50, 30);
  }
}
```

Una muestra de la salida es la siguiente:

A continuación se mostrará más detalladamente cómo funciona este applet. En primer lugar, como se esperaba, el applet **AppletDesplazarFrase** extiende **Applet**, pero también implementa a **Runnable**. Esto es necesario, ya que el applet va a crear un segundo hilo de ejecución que será utilizado para desplazar el mensaje. Dentro de **init( )**, se establecen los colores de fondo y de frente para el applet.

Después de la inicialización, el intérprete de Java llama a **start( )** para arrancar el applet. Dentro de **start( )**, se crea un hilo nuevo de ejecución que además se asigna a la variable **t** de tipo **Thread**. Después, la variable **stopFlag**, que es de tipo booleano que controla la ejecución del applet, se pone a **false**. A continuación, se llama a **t.start( )** con lo que se arranca el nuevo hilo. Hay que recordar

que **t.start( )** llama a un método definido por **Thread**, que causa que se ejecute el método **run( )** y no corresponde a una llamada del método **start( )** de la clase **Applet**, son dos métodos diferentes.

Dentro de **run( )**, los caracteres contenidos en **msg** se van moviendo continuamente hacia la izquierda. En cada movimiento se llama a **repaint( )**. Esto provoca que se llame cada vez al método **paint( )** y que se visualice **msg**. En cada iteración, **run( )** se duerme durante un cuarto de segundo. Lo que provoca **run( )** es que el contenido de **msg** sea movido y visualizado constantemente de derecha a izquierda. En cada iteración se revisa la variable **stopFlag**. Cuando ésta es **true**, el método **run( )** finaliza.

Si es un navegador el que visualiza el applet cuando se ve una nueva página, se llama al método **stop( ),** el cual pone **stopFlag** a **true**, haciendo que termine **run( )**. Éste es el mecanismo que se usa para detener un hilo si la página no se va a ver más. Cuando se vuelve a ver el applet, se vuelve a invocar a **start( )**, que arranca un nuevo hilo para ejecutar el movimiento de la frase.

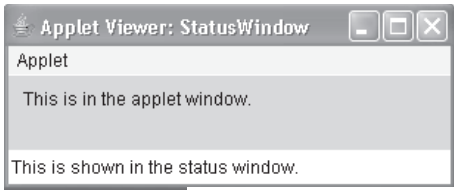
Uso de la barra de estado

Además de visualizar información en su ventana, un applet también puede mostrar un mensaje en la barra de estado del navegador o del visor de applets en el que se está ejecutando. Para ello hay que llamar al método **showStatus( )**, pasándole el texto que se desee mostrar. La barra de estado es un buen lugar para informar al usuario sobre lo que está ocurriendo en el applet, sugiriendo opciones, o informar de algún tipo de error. La barra de estado también sirve como una excelente ayuda para depurar, ya que es una vía por la que se puede sacar fácilmente información del applet.

En el siguiente applet se puede ver cómo trabaja **showStatus( )**:

```
// Usando la barra de estado.
import java.awt.*;
import java.applet.*;
/*
<applet code="BarraDeEstado" width=300 height=50>
</applet>
*/

public class BarraDeEstado extends Applet {
  public void init() {
    setBackground(Color.cyan);
  }

  // Muestra msg en la ventana del applet.
  public void paint(Graphics g) {
    g.drawString("Esto está en la ventana del applet.", 10, 20);
    showStatus("Esto se enseña en la barra de estado.");
  }
}
```

La siguiente figura es una muestra de la salida de este programa:



La etiqueta APPLET de HTML

Como se mencionó antes, Sun actualmente recomienda que la etiqueta APPLET sea utilizada para arrancar un applet tanto desde un documento HTML como desde un visor de applets. Un visor de applets ejecutará cada etiqueta APPLET que encuentre en una ventana independiente, mientras que los navegadores permiten la ejecución de varios applets en una misma página. Hasta aquí solamente se ha utilizado una forma simplificada de la etiqueta APPLET. Vamos ahora a profundizar en ella.

A continuación se muestra la sintaxis para una etiqueta APPLET completa. Lo que está entre corchetes es opcional.

```
< APPLET
  [CODEBASE = urlBaseCodigo]
  CODE = archivoClassApplet
  [ALT = textoAlternativo]
  [NAME = nombreInstanciaApplet]
  WIDTH = pixeles HEIGHT = pixeles
  [ALIGN = tipoAlineamiento]
  [VSPACE =pixeles] [HSPACE =pixeles]
>
[< PARAM NAME =NombreAtributo VALUE=ValorAtributo >]
[< PARAM NAME =NombreAtributo2 VALUE =ValorAtributo >]
…
[HTML que se visualiza en ausencia de Java]
</ APPLET>
```

Veamos cada parte.

CODEBASE CODEBASE es un atributo opcional que especifica el URL base del código del applet, esto es, el directorio en el que se buscará el archivo de clase ejecutable del applet (especificado por la etiqueta CODE). En caso de que no se haya especificado este atributo, se utiliza como CODEBASE el directorio URL del documento HTML. El CODEBASE no tiene por qué estar en el mismo equipo desde el que se lee el documento HTML.

CODE CODE es un atributo necesario y proporciona el nombre del archivo que contiene el archivo compilado **.class** del applet. Este archivo es ubicado con el URL base del applet, que es el directorio en el que está el archivo HTML o el directorio indicado en el propio atributo CODEBASE.

ALT La etiqueta ALT es un atributo opcional que especifica el mensaje que se debe visualizar en caso de que el navegador entienda la etiqueta APPLET pero no pueda ejecutar applets Java. Esto es diferente del HTML alternativo, que se proporciona para los navegadores que no admiten applets.

NAME NAME es un atributo opcional que se utiliza para especificar un nombre para una instancia del applet. Los applets deben ser nombrados para que otros applets de la misma página puedan encontrarlos por su nombre y puedan comunicarse con ellos. Para obtener un applet por su nombre, se utiliza el método **getApplet( )**, que se define mediante la interfaz **AppletContext**.

WIDTH Y HEIGHT WIDTH y HEIGHT son atributos necesarios que dan el tamaño del área de visualización de un applet (en píxeles).

ALIGN ALIGN es un atributo opcional que especifica la alineación del applet. Este atributo tiene el mismo tratamiento que en la etiqueta IMG de HTML; puede tomar los siguientes valores: LEFT, RIGHT, TOP, BOTTOM, MIDDLE, BASELINE, TEXTTOP, ABSMIDDLE, y ABSBOTTOM.

VSPACE Y HSPACE Estos atributos son opcionales. VSPACE especifica el espacio, en píxeles, por encima y por debajo del applet. HSPACE especifica el espacio, en píxeles, a cada lado del applet. Tienen el mismo tratamiento que los atributos VSPACE y HSPACE de la etiqueta IMG.

PARAM NAME Y VALUE La etiqueta PARAM permite definir argumentos específicos para applets en una página HTML. Los applets acceden a esos atributos con el método **getParameter( )**.

Otro atributo válido de la etiqueta APPLET es ARCHIVE, el cual permite especificar uno o más archivos comprimidos, y OBJECT, que especifica una versión almacenada del applet. En general, una etiqueta APPLET debe incluir sólo los atributos CODE u OBJECT, pero no ambos.

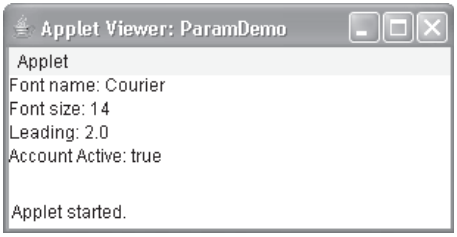
Paso de parámetros a los Applets

Como se acaba de ver, la etiqueta APPLET permite en HTML pasar parámetros a un applet. Para recuperar el valor de un parámetro, se utiliza el método **getParameter( )**. Éste devuelve el valor del parámetro como un objeto **String**. Por tanto, para valores numéricos y de tipo **boolean**, se necesitará convertir sus representaciones en texto a sus formatos internos. El siguiente es un ejemplo de cómo pasar parámetros:

```
// Uso de parámetros
import java.awt.*;
import java.applet.*;
/*
<applet code="ParamDemo" width=300 height=80>
<param name=nombreFuente value=Courier>
<param name=tamañoFuente value=l4>
<param name=separacion value=2>
<param name=accountEnabled value=true>
</applet>
*/

public class ParamDemo extends Applet{
  String nombreFuente;
  int tamañoFuente;
  float separacion;
  boolean active;

  // Inicializa el String que se va a visualizar.
  public void start() {
    String param;

    nombreFuente = getParameter("nombreFuente");
    if (nombreFuente == null)
      nombreFuente = "No encontrada";

    param = getParameter("tamañoFuente");
    try {
```

```
      if(param != null) // si se ha encontrado
        tamañoFuente= Integer.parseInt (param);
      else
        tamañoFuente = 0;
    } catch(NumberFormatException e) {
      tamañoFuente = -1;
    }

    param = getParameter("separacion");
    try {
      if(param != null) // si se ha encontrado
        separacion = Float.valueOf(param).floatValue();
      else
        separacion= 0;
    } catch(NumberFormatException e) {
      separacion = -1;
    }

    param = getParameter("accountEnabled");
    if(param != null)
      active = Boolean.valueOf(param).booleanValue();
  }

  // Mostrar los parámetros
  public void paint(Graphics g) {
    g.drawString("Nombre de la fuente: " + nombreFuente, 0, 10);
    g.drawString("Tamaño de la fuente: " + tamañoFuente, 0, 26);
    g.drawString("Separación: " + separación, 0, 42);
    g.drawString("Activo: " + active, 0, 58);
  }
}
```

A continuación se puede ver una muestra de la salida de este programa:



Como se ve en el programa, se deberían probar los valores que se obtienen con **getParameter( )**. Si un parámetro no está disponible, **getParameter( )** devolverá **null**. Además las conversiones a tipo numérico también se tienen que intentar en una sentencia **try** que gestione la excepción **NumberFormatException**. No deberá haber excepciones sin gestionar dentro de un applet.

Mejora del applet que muestra una frase

Se puede utilizar un parámetro para mejorar el applet visto anteriormente. En la versión anterior, el mensaje a mostrar por el applet estaba definido dentro del applet. Pero si pasamos el mensaje como un parámetro, esto permite al applet poder sacar un mensaje diferente cada vez que es

ejecutado. Esta nueva versión se muestra a continuación. Observe que la etiqueta APPLET al inicio del archivo especifica ahora un parámetro llamado **mensaje** que está relacionado con una cadena entre comillas.

```
// Un applet con parámetros
import java.awt.*;
import java.applet.*;
/*
<applet code="paramFrase" width=300 height=50>
<param name= mensaje value ="¡Java permite el movimiento en la Web!">
</applet>
*/

public class ParamFrase extends Applet inplements Runnable {
  String mensaje;
  Thread t = null;
  int estado;
  boolean stopFlag;

  // Establece colores e inicializa el hilo.
  public void init() {
    setBackground(Color.cyan);
    setForeground(Color.red);
  }

  // Comienza el hilo
  public void start() {
    mensaje = getParameter("mensaje");
    if(mensaje == null) mensaje = "Mensaje no encontrado";
    mensaje = " " + mensaje;
    t = new Thread(this);
    stopFlag = false;
    t.start();
  }

  // Punto de entrada del hilo que desplaza la frase.
  public void run() {
    char ch;

    // Visualiza la frase
    for( ; ; ) {
      try{
        repaint();
        Thread.sleep(250);
        ch =mensaje.charAt(0);
        mensaje = mensaje.substring(l, mensaje.length());
        mensaje += ch;
        if(stopF1ag)
          break;
      } catch(InterruptedException e) {}
    }
  }

  // Pausa en el desplazamiento de la frase.
  public void stop() {
    stopF1ag = true;
```

```
    t = null;
  }

  // Visualiza la frase.
  public void paint(Graphics g) {
    g.drawString(mensaje, 50, 30);
  }
}
```

getDocumentBase( ) y getCodeBase( )

A menudo, se crearán applets que necesiten explícitamente cargar texto e imágenes y/o sonido. Java permite al applet cargar datos desde el directorio en el que se encuentra el archivo HTML que lo arranca (conocido como *directorio base* del documento) y desde el directorio desde el que el archivo clase del applet fue cargado (directorio conocido como *base del código*). Estos directorios se devuelven como objetos **URL** (descritos en el Capítulo 20) llamando a los métodos **getDocumentBase( )** y **getCodeBase( )**. Se pueden concatenar con una cadena que contenga el nombre el archivo que se quiere cargar. Para cargar otro archivo, se utilizará el método **showDocument( )** definido por la interfaz **AppletContext**, que se verá más adelante.

Estos métodos se pueden ver en el siguiente applet:

```
// Muestra los directorios base de documento y código.
import java.awt.*;
import java.applet.*;
import java.net.*;
/*
<applet code="Bases" width=300 height=50>
</applet>
*/

public class Bases extends Applet{
  // Muestra los directorios base de documento y código.
  public void paint(Graphics g) {
    String msg;

    URL url = getCodeBase(); // obtiene la base de código
    msg = "Base de código: " + url.toString();
    g.drawString(msg, 10, 20);

    url = getDocumentBase(); // obtiene la base de documento
    msg = "Base de documento: " + url.toString();
    g.drawString(msg, 10, 40);
  }
}
```

A continuación se puede ver una muestra de la salida de este programa:

PARTE II

AppletContext y showDocument( )

Java puede utilizar imágenes activas y animación para dar un aspecto gráfico a la navegación en Internet, cosa que es mucho más interesante que las típicas palabras en azul subrayadas como hipertexto. Para permitir al applet transferir el control a otra URL, se debe utilizar el método **showDocument( )**, definido por la interfaz **AppletContext**. **AppletContext** es una interfaz que permite acceder a información del entorno de ejecución del applet. En la Tabla 21-2 se muestran los métodos definidos por **AppletContext**. El contexto del applet que se está ejecutando en un momento dado se obtiene llamando al método **getAppletContext( )** definido en la clase **Applet**.

Dentro de un applet, y una vez que se ha obtenido el contexto del applet, se puede visualizar otro documento llamando al método **showDocument( )**. Este método no devuelve ningún valor ni lanza ninguna excepción si falla, por lo que hay que utilizarlo con precaución. Hay dos métodos **showDocument( )**. El método **showDocument(URL)** visualiza el documento en la URL especificada. El método **showDocument(URL, String)** visualiza el documento en la posición especificada dentro de la ventana del navegador. Algunos argumentos válidos para la cadena que define la *posición* son "_self" (lo muestra en el marco actual), "_parent" (lo muestra en el marco padre), "_top" (lo muestra en el marco superior), y "_blank" (lo muestra en una nueva ventana del navegador). También se puede especificar un nombre, para que el documento se muestre en una nueva ventana del navegador con ese nombre.

El siguiente applet muestra **AppletContext** y **showDocument( )**. Tras la ejecución, se obtiene el contexto del applet actual y se utiliza para transferir el control a un archivo que se llama **Test.html**. Este archivo debe estar en el mismo directorio que el applet. **Test.html** puede contener cualquier hipertexto válido.

```
/* Utilizando un contexto de applet, getCodeBase()
y showDocument() para visualizar un archivo HTML.
*/

import java.awt.*;
import java.applet.*;
import java.net.*;
/*
<applet code="ACDemo" width=300 height=50>
</applet>
*/

public class ACDemo extends Applet{
  public void start() {
    AppletContext ac = getAppletContext();
    URL url = getCodeBase(); // obtiene el url de este applet

    try {
      ac.showDocument(new URL(url+"Test.html"));
    } catch{MalformedURLException e) {
      showStatus("URL no encontrado");
    }
  }
}
```

Método	Descripción
Applet getApplet(String *appletName*)	Devuelve el applet especificado por *appletName* si está dentro del contexto del applet actual. En caso contrario, devuelve **null**.
Enumeration<Applet> getApplets( )	Devuelve una enumeración con los applets que se encuentran dentro del contexto del applet actual.
AudioClip getAudioClip(URL *url*)	Devuelve un objeto **AudioClip** que encapsula el archivo de sonido encontrado en el sitio especificado por *url*.
Image getImage(URL *url*)	Devuelve un objeto **Image** que encapsula la imagen encontrada en el sitio especificada por *url*.
InputStream getStream(String *key*)	Devuelve el flujo relacionado a key. La cadena *key* fue enlazada con un flujo utilizando el método **setStream()**. Se devuelve **null** si ningún flujo está relacionado con key.
Iterator<String> getStreamKeys( )	Devuelve un iterador para los flujos asociados con el objeto invocante. Véase **getStream( )** y **setStream( )**.
void setStream(String *key*, InputStream *strm*)	Enlaza el flujo especificado por *strm* a la palabra especificada por *key*. *key* es eliminada del objeto invocante si strm es **null**.
void showDocument(URL *url*)	Visualiza el documento que se encuentra en el **URL** especificado por *url*. Este método no se puede utilizar cuando se trabaja con visores de applets.
void showDocument(URL *url*, String *where*)	Visualiza el documento que se encuentra en el **URL** especificado por *url*. Este método no se puede utilizar cuando se trabaja con visores de applets. La posición del documento es especificada por *where* como se describió antes.
void showStatus(String *str*)	Muestra la cadena *str* en la ventana de estado.

TABLA 21-2 Métodos abstractos definidos por la interfaz **AppletContext**

La interfaz AudioClip

La interfaz **AudioClip** define los siguientes métodos: **play( )** (reproduce un audio desde el principio), **stop( )** (detiene la reproducción del audio), y **loop( )** (reproduce un archivo continuamente). Después de cargar un archivo de audio utilizando **getAudioClip( )**, estos métodos se pueden utilizar para escucharlo.

La interfaz AppletStub

Mediante la interfaz **AppletStub**, se pueden comunicar un applet y el navegador (o el visor de applets). Normalmente, no se suele implementar esta interfaz en nuestros programas.

Salida a consola

Aunque la salida a la ventana de un applet debe ser realizada a través de métodos basados en interfaces gráficas, tal como **drawString( )**, es posible utilizar en el applet la salida a la consola, sobre todo para depurar. En un applet, cuando se llama a un método como **System.out.println( )**, la salida no se envía a la ventana del applet, sino que aparece ya sea en la sesión de consola en la que está el visor de applets, o bien en la consola de Java que está disponible en algunos navegadores. Se recomienda utilizar la salida a la consola únicamente para depurar, y seguir así los principios básicos de diseño que esperan la mayoría de los usuarios de interfaces gráficas.

CAPÍTULO 22

Gestión de eventos

Este capítulo examina un aspecto importante de Java: los eventos. La gestión de eventos es fundamental para la programación en Java porque se integra a la creación de applets y otros tipos de programas basados en interfaces gráficas. Como se explicó en el Capítulo 21, los applets son programas basados en eventos, que utilizan la interfaz gráfica de usuario, para interactuar con el usuario. Además, cualquier programa que utilice una interfaz gráfica de usuario, tal como una aplicación escrita en Java para Windows, se basa en eventos. Así que no se pueden escribir este tipo de programas sin un sólido dominio de manejo de eventos. Los eventos están soportados por un conjunto de paquetes, incluyendo **java.util**, **java.awt** y **java.awt.event**.

La mayoría de los eventos a los que un programa responderá son generados cuando el usuario interactúa con un programa basado en GUI. Este tipo de eventos son los que se examinan en este capítulo. Dichos eventos se pasan al programa de muchas formas diferentes, con el método específico dependiente del evento actual. Hay muchos tipos de eventos, incluidos aquellos que son generados por el ratón, el teclado, y otros controles GUI como los botones, scrollbar o checkbox.

Este capítulo comienza con una visión general del mecanismo de gestión de eventos de Java. Luego se examinan las interfaces y las clases de eventos principales utilizadas por el AWT y se desarrollan varios ejemplos que demuestran los fundamentos del procesamiento de eventos. En este capítulo también se explica cómo utilizar clases adaptadoras, clases internas y clases internas anónimas para estilizar el código al gestionar eventos. Los ejemplos que se muestran en el resto del libro hacen frecuentemente uso de estas técnicas.

NOTA *Este capítulo se enfoca en los eventos relacionados con los programas basados en GUI. Sin embargo, los eventos son también ocasionalmente utilizados para propósitos no directamente relacionados con programas basados en GUI. En todos los casos, las mismas técnicas básicas de gestión de eventos pueden ser aplicadas.*

Dos mecanismos para gestionar eventos

Antes de comenzar con la gestión de eventos, hay que dejar claro que la forma de gestionar los eventos ha cambiado significativamente entre la versión original de Java (1.0) y versiones posteriores de Java, comenzando por la versión 1.1. El modelo de gestión de eventos de la versión 1.0 todavía funciona, pero no se recomienda utilizarlo en programas nuevos. Además, muchos de los métodos que soportaba el antiguo modelo de eventos de la versión 1.0 se han quedado obsoletos. El nuevo modelo de gestión de eventos se debe utilizar en todos los programas nuevos, por ello será empleado por los programas en este libro.

El modelo de delegación de eventos

La propuesta actual de gestión de eventos se basa en lo que se llama *modelo de delegación de eventos*, el cual define mecanismos coherentes y estándares para generar y procesar eventos. Su concepto es muy sencillo: una *fuente* genera un *evento* y lo envía a uno o más *listeners*. En este esquema, los listeners simplemente esperan hasta que reciben un evento. Una vez recibido lo procesa y lo devuelve. La ventaja de este diseño es que la lógica de aplicación que procesa los eventos está claramente separada de la lógica de la interfaz de usuario que genera esos eventos. Un elemento de interfaz de usuario es capaz de"delegar"el procesamiento de un evento a una parte separada de código.

En el modelo de delegación de eventos, los listeners atienden a una fuente de la cual reciben la notificación de un evento. Esto da una ventaja importante: las notificaciones se envían sólo a los listeners que quieren recibirlas. Ésta es una forma más eficiente de gestionar eventos que el utilizado por el antiguo Java 1.0. Anteriormente, se propagaba un evento jerárquicamente hasta llegar a un componente que lo gestionaba. Esto requería componentes para recibir eventos que no iban a procesarse, lo cual significaba una pérdida de tiempo valioso. Con el modelo de delegación de eventos se evitan estos costos.

NOTA *Java también permite procesar eventos sin tener que utilizar el modelo de delegación de eventos. Se puede hacer extendiendo un componente AWT. Esta técnica se verá al final del Capítulo 24. Sin embargo, por las razones citadas anteriormente, el diseño preferido es el modelo de delegación de eventos.*

Las siguientes secciones definen los eventos y describen los roles de las fuentes y listeners.

Eventos

En el modelo de delegación, un *evento* es un objeto que describe un cambio de estado en una fuente. Se puede generar como consecuencia de que una persona interactúe con los elementos en una interfaz gráfica de usuario. Algunas de las actividades que causan la generación de eventos son presionar un botón, meter un carácter mediante el teclado, seleccionar un ítem de una lista, y hacer clic con el ratón, entre otros muchos.

Puede ocurrir que no se provoque un evento directamente por la interacción con una interfaz de usuario. Por ejemplo, se puede generar un evento cuando se termina un cronómetro, cuando un contador pasa de un cierto valor, cuando hay un fallo de software o hardware, o cuando se completa una operación. El programador es libre de definir los eventos que uno considere mejores para su aplicación.

Fuentes de eventos

Una *fuente* es un objeto que genera un evento. Esto ocurre cuando cambia de alguna manera el estado interno de ese objeto. Las fuentes pueden generar más de un tipo de eventos.

Una fuente tiene que registrar los listeners para que estos reciban notificaciones sobre un tipo específico de evento. Cada tipo de evento tiene su propio método de registro. La forma general es:

public void add*Type*Listener(*Type*Listener *el*)

Donde *Type* es el nombre del evento y *el* es una referencia al listener. Por ejemplo, el método que registra un evento de teclado a un listener es **addKeyListener( )**. El método que registra a un

listener de movimiento de ratón es **addMouseMotionListener( )**. Cuando ocurre un evento, se notifica a todos lo listeners registrados, y reciben una copia del objeto evento. Esto es lo que se conoce como *multidifusión* del evento. En todos los casos, las notificaciones se envían sólo a los listeners que quieren recibirlos.

Algunas fuentes sólo permiten registrar a un listener. La forma general del método de registro es:

```
public void add Type Listener(Type Listener el)
  throws java.util.TooManyListenersException
```

Donde *Type* es el nombre del evento y *el* es una referencia al listener. Cuando se produce tal evento, se notifica al listener que está registrado. Esto es conocido como *difusión única* del evento.

Una fuente también debe proporcionar un método que permita a un listener eliminar un registro en un tipo específico de evento. La forma general es:

```
public void remove Type Listener(Type Listener el)
```

Aquí, *Type* es el nombre del evento y *el* es una referencia al listener. Por ejemplo, para borrar un listener de teclado, se llamaría a **removeKeyListener( ).**

La fuente que genera eventos es la que proporciona los métodos para añadir o quitar listeners. Por ejemplo, la clase **Component** proporciona métodos para añadir o quitar listeners de eventos de teclado o ratón.

Auditores de eventos

Nota del traductor. Los auditores de eventos son mejor conocidos por su nombre en inglés **listener**. En el libro se mantendrá el uso de este término en inglés.

Un **listener** es un objeto que es notificado cuando ocurre un evento. Tiene dos requisitos principales. Primero, tiene que ser registrado con una o más fuentes para recibir notificaciones sobre eventos de tipos específicos. Segundo, tiene que implementar métodos para recibir y procesar esas notificaciones.

Los métodos que reciben y procesan eventos se definen en un conjunto de interfaces que están definidas en el paquete **java.awt.event.** Por ejemplo, la interfaz **MouseMotionListener** define dos métodos para recibir notificaciones cuando se arrastra o mueve el ratón. Cualquier objeto puede recibir y procesar uno de estos eventos, o ambos, si implementa esa interfaz. En este y otros capítulos se verán muchas más interfaces para **listeners**.

Clases de eventos

Las clases que representan eventos son el núcleo del mecanismo de gestión de eventos de Java. Así que una discusión sobre la gestión de eventos debe comenzar con la clases de eventos. Es importante comprender, sin embargo, que Java define varios tipos de eventos y que no todas las clases de eventos pueden ser discutidas en este capítulo. Los eventos más ampliamente utilizados son los definidos por ATW y los definidos por Swing. Este capítulo se enfoca en los eventos de AWT. La mayoría de estos eventos también aplican para Swing. Algunos eventos específicos de Swing se describen en el Capítulo 29.

Como raíz en la jerarquía de clases de eventos Java está la clase **EventObject**, definida en el paquete **java.util**. Ésta es la superclase para todos los eventos. Su constructor es el siguiente:

```
EventObject(Object src)
```

Donde *src* es el objeto que genera ese evento.

EventObject contiene dos métodos: **getSource( )** y **toString( )**. El método **getSource( )** devuelve la fuente del evento. Su forma general es la siguiente:

```
Object getSource( )
```

Como se esperaba, **toString( )** devuelve la cadena equivalente al evento.

La clase **AWTEvent**, definida dentro del paquete **java.awt**, es una subclase de **EventObject**. Esta es la superclase (directa o indirectamente) de todos los eventos basados en AWT utilizados por el modelo de delegación de eventos. Se puede utilizar el método **getID( )** para determinar el tipo del evento. La representación de este método se muestra aquí:

```
int getID( )
```

Al final del Capítulo 24 se verán más detalles sobre **AWTEvent**. Ahora sólo es importante conocer que las demás clases que se ven en este apartado son subclases de **AWTEvent**.

Como resumen:

- **EventObject** es la superclase de todos los eventos.
- **AWTEvent** es la superclase de todos los eventos AWT que se gestionan por medio del modelo de delegación de eventos.

El paquete **java.awt.event** define muchos tipos de eventos que se generan mediante elementos de interfaz de usuario. La Tabla 22-1 enumera las clases de eventos más importantes y describe brevemente cuándo se generan. Los constructores y métodos que se utilizan normalmente en cada clase se describen en apartados posteriores.

La clase ActionEvent

Un evento **ActionEvent** se genera cuando se presiona un botón, se hace doble clic en un elemento de una lista, o se selecciona un elemento de un menú. La clase **ActionEvent** define cuatro constantes enteras que se pueden utilizar para identificar cualquier modificador asociado con este tipo de evento: **ALT_MASK**, **CTRL_MASK**, **META_MASK** y **SHIFT_MASK**. Además existe una constante entera, **ACTION_PERFORMED**, que se puede utilizar para identificar eventos de acción.

Clase	Descripción
ActionEvent	Se genera cuando se presiona un botón, se hace doble clic en un elemento de una lista, o se selecciona un elemento de menú
AdjustmentEvent	Se genera cuando se manipula un scrollbar.
ComponentEvent	Se genera cuando un componente se oculta, se mueve, se cambia de tamaño, o se hace visible.
ContainerEvent	Se genera cuando se añade o se elimina un componente de un contenedor.
FocusEvent	Se genera cuando un componente gana o pierde el foco.
InputEvent	Superclase abstracta para cualquier clase de evento de entrada de componente.
ItemEvent	Se genera cuando se hace clic en un checkbox o en un elemento de una lista; también ocurre cuando se hace una selección en elemento de opción o cuando se selecciona o se deselecciona un elemento de un menú de opciones.

KeyEvent	Se genera cuando se recibe una entrada desde el teclado.
MouseEvent	Se genera cuando el ratón se arrastra, se mueve, se hace clic, se presiona, o se libera; también se genera cuando el ratón entra o sale de un componente.
MouseWheelEvent	Se genera cuando se mueve la rueda del ratón.
TextEvent	Se genera cuando se cambia el valor de un área de texto o un campo de texto.
WindowEvent	Se genera cuando una ventana se activa, se cierra, se desactiva, se minimiza, se maximiza, se abre, o se sale de ella.

TABLA 22-1 Principales clases de eventos en **java.awt.event**

La clase **ActionEvent** tiene estos tres constructores:

ActionEvent(Object *src*, int *tipo*, String *cmd*)
ActionEvent(Object *src*, int *tipo*, String *cmd*, int *modificadores*)
ActionEvent(Object *src*, int *tipo*, String *cmd*, long cuando, int *modificadores*)

Donde *src* es una referencia al objeto que ha generado este evento. El *tipo* del evento se especifica con tipo, y la cadena correspondiente a su comando es *cmd*. El argumento modificadores indica qué teclas modificadoras (ALT, CTRL, META y/o SHIFT) se han presionado cuando se ha generado el evento.Y el parámetro *cuando* especifica cuando ocurrió el evento.

Se puede obtener el nombre del comando del objeto **ActionEvent** invocado utilizando el método **getActionCommand( )**, como se muestra a continuación:

String getActionCommand( )

Por ejemplo, cuando se presiona un botón, se genera un evento de acción que tiene un nombre de comando igual a la etiqueta de ese botón.

El método **getModifiers( )** devuelve un valor que indica qué tecla modificadora (ALT, CTRL, META, y/o SHIFT) se ha presionado cuando el evento fue generado. Su forma es la siguiente:

int getModifiers( )

El método **getWhen( )** devuelve el tiempo en el cuál el evento tuvo lugar. Este dato es conocido como *la estampa en el tiempo* del evento. El método **getWhen( )** se muestra aquí:

long getWhen( )

La clase AdjustmentEvent

Se genera un objeto evento de la clase **AdjustmentEvent** como resultado de un cambio sobre una scrollbar. Existen cinco tipos de eventos del tipo **AdjustmentEvent**. La clase **AdjustmentEvent** define constantes enteras que se pueden utilizar para identificar dichos tipos. Las constantes y sus significados son los siguientes:

BLOCK_DECREMENT	El usuario hace clic dentro de la scrollbar para decrementar su valor.
BLOCK_INCREMENT	El usuario hace clic dentro de la scrollbar para incrementar su valor.
TRACK	Se arrastra el botón móvil de la scrollbar.
UNIT_DECREMENT	Se ha hecho clic en el botón que está al final de la scrollbar para decrementar su valor.

UNIT_INCREMENT	Se ha hecho clic en el botón que está al final de la scrollbar para incrementar su valor.

Además, hay una constante entera, **ADJUSTMENT_VALUE_CHANGED**, que indica que ha ocurrido un cambio.

AdjustmentEvent tiene el siguiente constructor:

AdjustmentEvent(Adjustable *src*, int *id*, int *tipo*, int *data*)

Aquí, *src* es una referencia al objeto que ha generado ese evento. El *id* específica el evento. El tipo del ajuste es especificado por *tipo*, y sus datos asociados están en *data*.

El método **getAdjustable( )** devuelve el objeto que ha generado el evento. Su forma es la siguiente:

Adjustable getAdjustable( )

El tipo de evento de ajuste se puede obtener mediante el método **getAdjustmentType( )**, que devuelve una de las constantes definidas por **AdjustmentEvent**. La forma general es la siguiente:

int getAdjustmentType( )

La cantidad del ajuste se puede obtener del método **getValue( )**, que se muestra a continuación:

int getValue( )

Por ejemplo, cuando se manipula una scrollbar, este método devuelve el valor representado por ese cambio.

La clase ComponentEvent

Se genera un objeto evento de la clase **ComponentEvent** cuando cambia el tamaño, posición o visibilidad de un componente. Hay cuatro tipos de eventos generados por un componente. La clase **ComponentEvent** define constantes enteras que se pueden utilizar para identificarlos. Las constantes y sus significados son las siguientes:

COMPONENT_HIDDEN	Se ha ocultado el componente.
COMPONENT_MOVED	Se ha movido el componente.
COMPONENT_RESIZED	Se ha cambiado el tamaño del componente.
COMPONENT_SHOWN	El componente se ha hecho visible.

ComponentEvent tiene este constructor:

ComponentEvent(Component *src*, int *tipo*)

Donde *src* es una referencia al objeto que ha generado el evento. El tipo del evento es especificado por *tipo*.

ComponentEvent es la superclase directa o indirectamente, de **ContainerEvent**, **FocusEvent**, **KeyEvent**, **MouseEvent** y **WindowEvent**.

El método **getComponent( )** devuelve el componente que ha generado el evento, como se ve a continuación:

Component getComponent( )

La clase ContainerEvent

Se genera un objeto evento de la clase **ContainerEvent** cuando se añade o se borra un componente desde un contenedor. Hay dos tipos de eventos generados por un contenedor. La clase **ContainerEvent** define constantes enteras que se pueden utilizar para identificarlos: **COMPONENT_ADDED** y **COMPONENT_REMOVED**. Éstas indican que se ha añadido o se ha borrado, respectivamente, un componente desde el contenedor.

ContainerEvent es una subclase de **ComponentEvent** y tiene el siguiente constructor:

ContainerEvent(Component *src*, int *tipo*, Component *comp*)

Donde *src* es una referencia al contenedor que genera ese evento. El tipo del evento es especificado por *tipo*, y el componente que se ha añadido o se ha borrado desde el contenedor es referenciado por *comp*.

Utilizando el método **getContainer( )**, se puede obtener una referencia al contenedor que ha generado el evento:

Container getContainer( )

El método **getChild( )** devuelve una referencia al componente que se ha añadido o se ha borrado desde el contenedor. Su forma general es la siguiente:

Component getChild( )

La clase FocusEvent

Se genera un objeto evento de la clase **FocusEvent** cuando un componente está o deja de estar activo. Estos eventos se identifican mediante las constantes enteras **FOCUS_GAINED** y **FOCUS_LOST**.

FocusEvent es una subclase de **ComponentEvent** y tiene estos constructores:

FocusEvent(Component *src*, int *tipo*)
FocusEvent(Component *src*, int *tipo*, boolean *temporalBandera*)
FocusEvent(Component *src*, int *type*, boolean *temporalBandera*, Component *otro*)

Donde *src* es una referencia al componente que ha generado este evento. El tipo del evento es especificado por *tipo*. El argumento *temporalBandera* es puesto en **verdadero** si el evento foco es temporal. En cualquier otro caso, se pone a **falso**. Un evento foco temporal se produce como resultado de otra operación de interfaz de usuario. Por ejemplo, supongamos que lo que está activo —tiene el foco— es un campo de texto. Si el usuario mueve el ratón para modificar una scrollbar, se pierde temporalmente el foco.

El otro componente envuelto en los cambios de foco, llamado el *componente opuesto*, se pasa en el argumento *otro*. Por lo tanto, si un evento **FOCUS_GAINED** ocurre, *otro* hará referencia al componente que perdió el foco. Por el contrario, si un evento **FOCUS_LOST** ocurre, *otro* hará una referencia al componente que obtuvo el foco.

Se puede determinar el componente otro llamando al método **getOppositeComponent( )**, que se muestra aquí:

Component getOppositeComponent( )

Devuelve el componente opuesto.

El método **isTemporary( )** indica si este cambio de foco es temporal. Su forma es la siguiente:

boolean isTemporary( )

El método devuelve **verdadero** si el cambio es temporal. Si no, devuelve **falso**.

La clase InputEvent

La clase abstracta **InputEvent** es una subclase de **ComponentEvent** y es la superclase para los eventos de entrada de componentes. Sus subclases son **KeyEvent** y **MouseEvent**.

La clase **InputEvent** define varias constantes enteras que representan a sus modificadores, por ejemplo presionar la tecla CTRL. Originalmente la clase **InputEvent** definía los siguientes ocho valores para definir a sus modificadores:

ALT_MASK	BUTTON2_MASK	META_MASK
ALT_GRAPH_MASK	BUTTON3_MASK	SHIFT_MASK
BUTTONI_MASK	CTRL_MASK	

Sin embargo, a causa de posibles conflictos entre los modificadores utilizados por eventos del teclado y ratón con otros eventos, los siguientes valores de modificadores extendidos fueron agregados:

ALT_DOWN_MASK	BUTTON2_DOWN_MASK	META_DOWN_MASK
ALT_GRAPH_DOWN_MASK	BUTTON3_DOWN_MASK	SHIFT_DOWN_MASK
BUTTON1_DOWN_MASK	CRTL_DOWN_MASK	

Cuando se escribe un código nuevo, se recomienda que se utilicen los nuevos modificadores extendidos en lugar de los modificadores originales.

Para verificar si un modificador fue presionado en el momento en que un evento se genera, se utilizan los métodos **isAltDown( )**, **isAltGraphDown( )**, **isControlDown( )**, **isMetaDown( )** y **isShiftDown( )**. Las formas de estos métodos son las siguientes:

```
boolean isAltDown( )
boolean isAltGraphDown( )
boolean isControlDown( )
boolean isMetaDown( )
boolean isShiftDown( )
```

El método **getModifiers( )** devuelve un valor que contiene todas las etiquetas de los modificadores para ese evento:

```
int getModifiers( )
```

Se pueden obtener los modificadores extendidos llamando **getModifiersEx( )**, el cual se muestra a continuación:

```
int getModifiersEx( )
```

La clase ItemEvent

Se genera un objeto evento de la clase **ItemEvent** cuando se hace clic en un checkbox o en un elemento de una lista o cuando se selecciona o se deselecciona un elemento de un menú de opciones. Los checkboxes y listboxes se describen más adelante. Hay dos tipos de eventos de elemento, que se identifican por las siguientes constantes enteras:

DESELECTED	El usuario deselecciona un elemento.
SELECTED	El usuario selecciona un elemento.

Además, **ItemEvent** define una constante entera, **ITEM _STATE_CHANGED**, que significa un cambio de estado.

ItemEvent tiene el siguiente constructor:

ItemEvent(ItemSelectable *src*, int *tipo*, Object *entrada*, int *estado*)

Donde *src* es una referencia al componente que ha generado ese evento. Un ejemplo podría ser una lista o un elemento de elección. El tipo de elemento es especificado por *tipo*. Lo que específicamente genera el evento de elemento se pasa con *entrada*. El estado actual del elemento es *estado*.

El método **getItem( )** se puede utilizar para obtener una referencia al elemento que ha generado un evento. Su forma es la siguiente:

Object getItem( )

El método **getItemSelectable( )** se puede utilizar para obtener una referencia al objeto **ItemSelectable** que ha generado el evento. Su forma general es la siguiente:

ItemSelectable getItemSelectable( )

Las listas y las listas desplegables son ejemplos de elementos de interfaz de usuario que implementan la interfaz **ItemSelectable**.

El método **getStateChange( )** devuelve el cambio de estado (por ejemplo, **SELECTED** o **DESELECTED**) para el evento. Su forma general es la siguiente:

int getStateChange( )

La clase KeyEvent

Se genera un objeto evento de la clase **KeyEvent** cuando se pulsa una tecla. Hay tres clases de eventos de teclado, que están definidos por las siguientes constantes enteras: **KEY_PRESSED**, **KEY_RELEASED**, y **KEY_TYPED**. Los primeros dos eventos se generan cuando se presiona o se libera cualquier tecla. El último evento sólo se da cuando se genera un carácter. Recordemos que no siempre se generan caracteres al presionar teclas. Por ejemplo, si se presiona la tecla SHIFT no se genera carácter alguno.

KeyEvent define otras muchas constantes enteras. Por ejemplo, **VK_0** a **VK_9** y **VK_A** a **VK_Z** definen los equivalentes ASCII de números y letras. Algunas otras constantes son:

VK_ALT	VK_DOWN	VK_LEFT	VK_RIGHT
VK_CANCEL	VK_ENTER	VK_ PAGE_DOWN	VK_SHIFT
VK_CONTROL	VK_ESCAPE	VK_PAGE_UP	VK_UP

Las constantes **VK** especifican *códigos de teclas virtuales* y son independientes de cualquier modificador, como control, shift o alt.

KeyEvent es una subclase de **InputEvent** y éste es uno de sus constructores:

KeyEvent(Component *src*, int *tipo*, long *cuando*, int *modificadores*, int *codigo*, char *ch*)

Donde *src* es una referencia al componente que genera ese evento. El tipo del evento es especificado por *tipo*. El tiempo en el que se ha presionado la tecla se pasa con *cuando*. El argumento *modificadores* indica qué modificador se ha presionado cuando ha ocurrido ese evento de teclado. Los códigos de tecla virtual, como **VK_UP**, **VK_A**, y demás, se pasan en el argumento llamado *codigo*. El carácter equivalente (si existe alguno) se pasa en *ch*. Si no existe

ningún carácter válido, entonces *ch* contiene **CHAR_UNDEFINED**. Para los eventos **KEY_TYPED**, el argumento *codigo* contendrá a **VK_UNDEFINED**.

La clase **KeyEvent** define varios métodos, pero los que más se usan son **getKeyChar( )**, que devuelve el carácter que se ha tecleado y **getKeyCode( )**, que devuelve el código de la tecla. Sus formas generales son las siguientes:

```
char getKeyChar( )
int getKeyCode( )
```

Si ningún carácter válido está disponible, entonces **getKeyChar( )** devuelve **CHAR_UNDEFINED**. Cuando se produce un evento **KEY_TYPED**, **getKeyCode( )** devuelve **VK_UNDEFINED**.

La clase MouseEvent

Existen ocho tipos de eventos de ratón. La clase **MouseEvent** define las siguientes constantes enteras, que se pueden utilizar para identificarlos:

MOUSE_CLICKED	El usuario hace clic con el ratón.
MOUSE_DRAGGED	El usuario arrastra el ratón.
MOUSE_ENTERED	El ratón entra a un componente.
MOUSE_EXITED	El ratón sale de un componente.
MOUSE_MOVED	Se mueve el ratón.
MOUSE_PRESSED	Se presiona el ratón.
MOUSE_RELEASED	Se libera el ratón.
MOUSE_WHEEL	La rueda del ratón fue movida

MouseEvent es una subclase de **InputEvent** y tiene este constructor:

```
MouseEvent(Component src, int tipo, long cuando, int modificadores,
int x, int y, int clics, boolean t)
```

Donde *src* es una referencia al componente que ha generado el evento. El tipo del evento es especificado por *tipo*. El tiempo al momento en el que ha ocurrido el evento de ratón se pasa en el argumento llamado *cuando*. El argumento *modificadores* indica qué modificador se ha presionado cuando ha ocurrido el evento. Las coordenadas del ratón se pasan con *x y*. El número de clics se pasa en *clics*. La etiqueta *t* indica si ese evento hace que aparezca un menú en esa plataforma.

Dos métodos comúnmente utilizados de esta clase son **getX( )** y **getY( )**. Estos métodos devuelven las coordenadas X,Y del ratón dentro de un componente cuando ha ocurrido el evento. Sus formas son las siguientes:

```
int getX( )
int getY( )
```

También se puede utilizar el método **getPoint( )** para obtener las coordenadas del ratón, como se muestra a continuación:

```
Point getPoint( )
```

Este método devuelve un objeto **Point** que contiene las coordenadas X,Y en sus miembros enteros: **x**, **y**. El método **translatePoint( )** traduce la posición del evento. Su forma es la siguiente:

void translatePoint(int *x*, int *y*)

Aquí, los argumentos *x*, *y* se añaden a las coordenadas del evento.

El método **getClickCount( )** proporciona el número de clics que se han hecho con el ratón para este evento. Su forma es la siguiente:

int getClickCount( )

El método **isPopupTrigger( )** prueba si el evento ha hecho aparecer un menú en la plataforma. Su forma es la siguiente:

boolean isPopupTrigger( )

También está disponible el método **getButton( )**, que se muestra a continuación

int getButton( )

Dicho método devuelve el valor que representa el botón que causa el evento. El valor devuelto será una de las constantes definidas por **MouseEvent**

NOBUTTON	BUTTON1	BUTTON2	BUTTON3

El valor de **NOBUTTON** indica que no hay botón presente o liberado.

Java SE 6 agrega tres métodos a **MouseEvent** que obtienen las coordenadas del ratón relativas a la pantalla en lugar de al componente. Estos métodos se muestran aquí:

Point getLocationOnScreen( )

int getXOnScreen( )

int getYOnScreen( )

El método **getLocationOnScreen( )** devuelve un objeto **Point** que contiene las dos coordenadas X y Y. Los otros dos métodos devuelven la coordenada correspondiente.

La clase MouseWheelEvent

La clase **MouseWheelEvent** encapsula un evento de la rueda del ratón. Es una subclase de **MouseEvent**. No todos los ratones tienen rueda. Si el ratón tiene rueda, ésta se encuentra entre los botones derecho e izquierdo. La rueda se utiliza para desplazarse. **MouseWheelEvent** define estas dos constantes enteras:

WHEEL_BLOCK_SCROLL	Un evento de página arriba o página abajo ocurrió
WHEEL_UNIT_SCROLL	Un evento de línea arriba o línea abajo ocurrió

A continuación uno de los constructores definidos por **MouseWheelEvent**:

MouseWheelEvent (Component *src*, int *tipo*, long *cuando*, int *modificadores*,
int *x*, int *y*, int *clics*, boolean *t*,
int *sc*, int *monto*, int *cuenta*)

Donde, *src* es una referencia al objeto que genera el evento. El tipo de evento se especifica en *tipo*. La hora en que el evento del ratón ocurrió se pasa en *cuando*. El argumento modificadores indica cuales modificadores fueron activados cuando el evento ocurrió. Las coordenadas del ratón se pasan a través de *x*, *y*. El número de clics que la rueda ha rotado se pasa mediante *clics*.

El argumento *t* es una bandera que indica si este evento causa que un menú popup aparezca en esta plataforma. El valor *sc* debe ser **WHEEL_UNIT_SCROLL** o **WHEEL_BLOCK_SCROLL**. El número de unidades de scroll es pasado en *monto*. El parámetro *cuenta* indica el número de rotaciones en que la rueda se movió.

MouseWheelEvent define métodos que permiten acceder a los eventos de la rueda. Para obtener el número de unidades rotacionales, se llama a **getWheelRotation( )**, como se muestra aquí:

```
int getWheelRotation( )
```

Este método devuelve el número de unidades rotacionales. Si el valor es positivo, la rueda se movió en contra de las manecillas del reloj. Si el valor es negativo, la rueda se movió en el sentido de las manecillas del reloj.

Para obtener el tipo de scroll, se llama a **getScrollType( )**, como se muestra aquí:

```
int getScrollType( )
```

que devuelve **WHEEL_UNIT_SCROLL** o **WHEEL_BLOCK_SCROLL**.

Si el tipo de scroll es **WHEEL_UNIT_SCROLL**, se puede obtener el número de unidades de scroll llamando al método **getScrollAmount( )**. Como se muestra aquí;

```
int getScrollAmount( )
```

La clase TextEvent

Con esta clase se describen eventos de texto. Se generan mediante campos de texto y áreas de texto cuando el usuario o el programa introducen caracteres. **TextEvent** define la constante entera **TEXT_VALUE_CHANGED**.

El único constructor para esta clase es el siguiente:

```
TextEvent(Object src, int tipo)
```

Donde *src* es una referencia al objeto que ha generado el evento. El tipo de evento es especificado por *tipo*.

El objeto **TextEvent** no incluye los caracteres que ya están en el componente de texto que ha generado el evento, sino que es el programa el que debe de utilizar otros métodos asociados con el componente de texto para recuperar la información. Esta operación es diferente a la de otros objetos de evento que se ven en esta sección. Por esta razón, no se estudia aquí ningún método para la clase **TextEvent**. Hay que pensar en una notificación de evento de texto como una señal a un listener para que este recupere información desde un componente de texto específico.

La clase WindowEvent

Hay diez tipos de eventos de ventana. La clase **WindowEvent** define constantes enteras que se pueden utilizar para identificarlos. Las constantes y sus significados son los siguientes:

WINDOW _ACTIVATED	Se ha activado la ventana.
WINDOW _CLOSED	Se ha cerrado la ventana.
WINDOW _CLOSING	El usuario ha pedido que se cierre la ventana.
WINDOW _DEACTIVATED	La ventana ha dejado de estar activa.
WINDOW_DEICONIFIED	Se ha mostrado la ventana tras pulsar su incono.
WINDOW_GAINED_FOCUS	La ventana ahora está en foco de entrada

WINDOW_ICONIFIED	Se ha minimizado la ventana a un icono.
WINDOW_LOST_FOCUS	La ventana ha perdido el foco de entrada.
WINDOW _OPENED	La ventana ha sido abierta.
WINDOW_STATE_CHANGED	El estado de la ventana ha cambiado.

WindowEvent es una subclase de **ComponentEvent** y define varios constructores. El primero se muestra aquí:

WindowEvent(Window *src*, int *tipo*)

Donde *src* es una referencia al objeto que ha generado el evento. El tipo del evento es *tipo*.

Los tres constructores siguientes ofrecen un control más detallado:

WindowsEvent(Windows *src*, int *tipo*, Window *otro*)
WindowsEvent(Windows *src*, int *tipo*, int *deEstado*, in *aEstado*)
WindowsEvent(Windows *src*, int *tipo*, Window *otro*, int *deEstado*, int *aEstado*)

Donde, *otro* especifica la ventana opuesta cuando un evento de foco o activación ocurre. El parámetro *deEstado* especifica el estado anterior de la ventana, y *aEstado* especifica el nuevo estado que la ventana tendrá cuando un cambio de estado de ventana ocurre.

El método más comúnmente utilizado de esta clase es **getWindow( )**, que devuelve el objeto **Window** que ha generado el evento. Su forma general es la siguiente:

Window getWindow( )

WindowEvent también define métodos que devuelven la ventana contraria (cuando un evento de foco o activación ha ocurrido), el estado anterior de la ventana, y el estado actual de la ventana. Estos métodos se muestran aquí:

Window getOppositeWindow( )
int getOldState( )
int getNewState( )

Fuentes de eventos

La Tabla 22-2 lista algunos de los componentes de interfaz de usuario que pueden generar los eventos descritos en el apartado anterior. Además de esos elementos de interfaz gráfica de usuario, cualquier clase derivada de **Component**, como **Applet**, puede generar eventos.

Fuente del evento	Descripción
Button	Genera eventos de tipo ActionEvent cuando se presiona el botón.
Checkbox	Genera eventos de tipo ItemEvent cuando se selecciona o se deselecciona un checkbox.
Choice	Genera eventos de tipo ItemEvent cuando se cambia una opción.
List	Genera eventos de tipo ActionEvent cuando se hace doble clic sobre un elemento; genera eventos de tipo ItemEvent cuando se selecciona o se deselecciona un elemento.

TABLA 22-2 Ejemplos de componentes que pueden generar eventos

Fuente del evento	Descripción
MenuItem	Genera eventos de tipo ActionEvent cuando se selecciona un elemento de menú; genera eventos de tipo ItemEvent cuando se selecciona o se deselecciona un elemento de un menú de opciones.
Scrollbar	Genera eventos de tipo AdjustmentEvent cuando se manipula el scrollbar.
Componentes de tipo Text	Genera eventos de tipo TextEvent cuando el usuario introduce un carácter.
Window	Genera eventos de tipo WindowEvent cuando una ventana se activa, se cierra, se desactiva, se minimiza, se maximiza, se abre o se sale de ella.

TABLA 22-2 Ejemplos de componentes que pueden generar eventos (*Continuación*)

Por ejemplo, se pueden recibir eventos del ratón y del teclado desde un applet. También el programador puede construir sus propios componentes que generen eventos. En este capítulo sólo se van a gestionar eventos de ratón y de teclado, pero en los dos siguientes capítulos se verá cómo gestionar los eventos que aparecen en la Tabla 22-2.

Las interfaces de auditores de eventos

Como se explicó anteriormente, el modelo de delegación de eventos tiene dos partes: fuentes y listeners. Los listeners se crean implementando una o más interfaces definidas en el paquete **java.awt.event**. Cuando se produce un evento, la fuente del evento invoca al método apropiado definido por el listener y proporciona un objeto evento como su argumento. En la Tabla 22-3 aparecen las interfaces de listener que más se utilizan, y también aporta una breve descripción de los métodos que definen esas interfaces. En las siguientes secciones se examinan los métodos específicos que hay en cada interfaz.

La interfaz ActionListener

Esta interfaz define el método **actionPerformed( )** que se invoca cuando un evento de acción ocurre. Su forma general se muestra a continuación:

```
void actionPerformed(ActionEvent ae)
```

Interfaz	Descripción
ActionListener	Define un método para recibir eventos de acción.
AdjustmentListener	Define un método para recibir eventos de ajuste.
ComponentListener	Define cuatro métodos para reconocer cuándo se oculta, se mueve, se cambia de tamaño o se muestra un componente.
ContainerListener	Define dos métodos para reconocer cuándo se añade o se elimina un componente de un contenedor.
FocusListener	Define dos métodos para reconocer cuándo un componente gana o pierde el foco del teclado.
ItemListener	Define un método para reconocer cuándo cambia el estado de un elemento.

KeyListener	Define tres métodos para reconocer cuándo se presiona, se libera o se golpea una tecla.
MouseListener	Define cinco métodos para reconocer cuándo se presiona o se libera un botón del ratón, se hace clic con él, o el ratón entra en un componente o sale de él.
MouseMotionListener	Define dos métodos para reconocer cuándo se arrastra o se mueve el ratón.
MouseWheelListener	Define un método para reconocer cuando la rueda del ratón se mueve.
TextListener	Define un método para reconocer cuándo cambia un valor de texto.
WindowFocusListener	Define dos métodos para reconocer cuando una ventana gana o pierde foco de entrada.
WindowListener	Define siete métodos para reconocer cuándo una ventana se activa, se cierra, se desactiva, se minimiza, se maximiza, se abre o se sale de ella.

TABLA 22-3 Interfaces de listener que más se utilizan

La interfaz AdjustmentListener

Esta interfaz define el método **adjustmentValueChanged( )** que se invoca cuando se produce un evento de ajuste. Su forma general es la siguiente:

```
void adjustmentValueChanged(AdjustmentEvent ae)
```

La interfaz ComponentListener

Esta interfaz define cuatro métodos que se invocan cuando a un componente se le cambia el tamaño, se mueve, se muestra o se oculta. Sus formas generales son las siguientes:

```
void componentResized(ComponentEvent ce)
void componentMoved(ComponentEvent ce)
void componentShown(ComponentEvent ce)
void componentHidden(ComponentEvent ce)
```

La interfaz ContainerListener

Esta interfaz tiene dos métodos. Cuando se añade un componente a un contenedor, se invoca a **componentAdded( )**. Cuando se borra un componente de un contenedor, se invoca a **componentRemoved( )**. Sus formas generales son las siguientes:

```
void componentAdded(ContainerEvent ce)
void componentRemoved(ContainerEvent ce)
```

La interfaz FocusListener

Esta interfaz define dos métodos. Cuando un componente obtiene el foco del teclado, se invoca **focusGained( )**. Cuando un componente pierde el foco del teclado, se llama a **focusLost( )**. Sus formas generales son las siguientes:

```
void focusGained(FocusEvent fe)
void focusLost(FocusEvent fe)
```

La interfaz ItemListener

Esta interfaz define el método **itemStateChanged( )** que se invoca cuando cambia el estado de un elemento. Su forma general es la siguiente:

void itemStateChanged(ItemEvent *ie*)

La interfaz KeyListener

Esta interfaz define tres métodos. Los métodos **keyPressed( )** y **keyReleased( )** se invocan cuando se presiona y se libera una tecla, respectivamente. El método **keyTyped( )** se invoca cuando se ha introducido un carácter.

Por ejemplo, si un usuario presiona y libera la tecla A, se generan tres eventos en secuencia: tecla presionada, carácter introducido, tecla liberada. Si un usuario presiona y libera la tecla INICIO, se generan dos eventos en este orden: tecla presionada, tecla liberada.

Las formas generales de estos eventos son las siguientes:

void keyPressed(KeyEvent *ke*)
void keyReleased(KeyEvent *ke*)
void keyTyped(KeyEvent *ke*)

La interfaz MouseListener

Esta interfaz define cinco métodos. Si se presiona y se libera el ratón en el mismo punto, se invoca al método **mouseClicked( )**. Cuando el ratón entra en un componente, se llama al método **mouseEntered( )**. Cuando el ratón sale del componente, se llama a **mouseExited( )**. Los métodos **mousePressed( )** y **mouseReleased( )** se invocan cuando se presiona y se libera el ratón, respectivamente. Las formas generales de estos métodos son las siguientes:

void mouseClicked(MouseEvent *me*)
void mouseEntered(MouseEvent *me*)
void mouseExited(MouseEvent *me*)
void mousePressed(MouseEvent *me*)
void mouseReleased(MouseEvent *me*)

La interfaz MouseMotionListener

Esta interfaz define dos métodos. Al método **mouseDragged( )** se le llama tantas veces como se arrastre al ratón. Al método **mouseMoved( )** se le llama tantas veces como se mueva el ratón. Sus formas generales son las siguientes:

void mouseDragged(MouseEvent me)
void mouseMoved(MouseEvent me)

La interfaz MouseWheelListener

Esta interfaz define el método **mouseWheelMoved( )** que se invoca cuando la rueda del ratón se mueve. Su forma general es:

void mouseWheelMoved(MouseWheelEvent mwe)

La interfaz TextListener

Esta interfaz define el método **textChanged( )** que se invoca cuando hay un cambio en una área de texto o en un campo de texto. Su forma general es la siguiente:

void textChanged(TextEvent *te*)

La interfaz WindowFocusListener

Esta interfaz define dos métodos: **windowGainedFocus( )** y **windowLostFocus( )**. Estos métodos son llamados cuando una ventana gana o pierde el foco de entrada. Sus formas generales se muestran a continuación:

void windowGainedFocus(WindowEvent *we*)
void windowLostFocus(WindowEvent *we*)

La interfaz WindowListener

Esta interfaz define siete métodos. Los métodos **windowActivated( )** y **windowDeactivated( )** se invocan cuando se activa o desactiva una ventana, respectivamente. Si una ventana se minimiza a un icono, se llama al método **windowIconified( )**. Cuando una ventana es mostrada pulsando en su icono, se llama al método **windowDeiconified( )**. Cuando se abre o se cierra una ventana, se llama a los métodos **windowOpened( )** o **windowClosed( )**, respectivamente. Al método **windowClosing( )** se le llama cuando se está cerrando una ventana. Las formas generales de estos métodos son:

void windowActivated(WindowEvent *we*)
void windowClosed(WindowEvent w*e*)
void windowClosing(WindowEvent *we*)
void windowDeactivated(WindowEvent *we*)
void windowDeiconified(WindowEvent *we*)
void windowIconified(WindowEvent *we*)
void windowOpened(WindowEvent *we*)

Uso del modelo de delegación de eventos

Una vez que se ha visto la teoría sobre la que trabaja el método de delegación de eventos y se tiene una idea general de sus componentes, se puede comenzar a trabajar con él. Utilizar el método de delegación de eventos es bastante fácil. Sólo hay que seguir estos dos pasos:

1. Implementar la interfaz apropiada en el listener, de tal manera que reciba el tipo de evento deseado.
2. Implementar el código para registrar y eliminar (si fuese necesario) el registro del listener como destinatario de las notificaciones de eventos.

Hay que recordar que una fuente puede generar muchos tipos de eventos. Cada evento se tiene que declarar de forma separada. Asimismo, se puede registrar un objeto para que reciba varios tipos de eventos, pero se deben implementar todas las interfaces que hagan falta para recibir esos eventos.

Para ver cómo trabaja el método de delegación de eventos, vamos a ver algunos ejemplos donde se gestionan los dos generadores de eventos que más se utilizan: el ratón y el teclado.

La gestión de eventos de ratón

Para gestionar eventos de ratón, se deben implementar las interfaces **MouseListener** y **MouseMotionListener** (posiblemente se desee también implementar **MouseWheelListener**, pero no se hará aquí). El siguiente applet muestra el proceso. Se van a visualizar las coordenadas

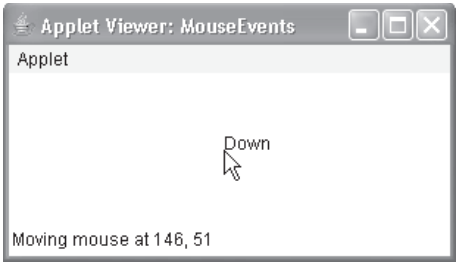
actuales del ratón en la ventana de estado del applet. Cada vez que se presione un botón, se verá la palabra "Abajo" en la posición que apunta el ratón. Y cuando se libere el botón, aparecerá la palabra "Arriba". Si se hace clic en un botón, se verá el mensaje "Clic del ratón" en la esquina superior izquierda del panel del applet.

Cuando el ratón entre o salga de la ventana del applet, se verá un mensaje en la esquina superior izquierda del panel del applet. Cuando se arrastre el ratón, se mostrará un *, que se irá dejando pista de por dónde se ha ido arrastrando el ratón. Hay que tener en cuenta que las dos variables, **mouseX** y **mouseY**, guardan la posición del ratón cuando se da un evento presionar, liberar o arrastrar el ratón. Estas coordenadas las utiliza **paint( )** para pintar la salida en el punto en que se han dado esos eventos.

```
// Ejemplo de la gestión de eventos de ratón.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code = "MouseEvent" width=300 height=100>
</applet>
*/

public c1ass MouseEvents extends Applet
  implements MouseListener, MouseMotionListener {

  String msg ="";
  int mouseX = 0, mouseY = 0; // coordenadas del ratón

  public void init() {
    addMouseListener(this);
    addMouseMotionListener(this);
  }

  // Gestiona los clics del ratón.
  public void mouseClicked(MouseEvent me) {
    // Guarda coordenadas
    mouseX = 0;
    mouseY = 10;
    msg = "Clic del ratón";
    repaint();
  }

  // Gestiona entradas del ratón
  public void mouseEntered(MouseEvent me) {
    // Guarda coordenadas
    mouseX = 0;
    mouseY = 10;
    msg = "El ratón ha entrado.";
    repaint();
  }

  // Gestiona salidas del ratón
  public void mouseExited(MouseEvent me) {
    // Guarda las coordenadas
    mouseX = 0;
    mouseY = 10;
    msg = "El ratón ha salido.";
    repaint() ;
  }
```

```
  // Gestiona presionar los botones.
  public void mousePressed(MouseEvent me) {
    // Guarda las coordenadas
    mouseX = me.getX();
    mouseY = me.getY() ;
    msg = "Abajo";
    repaint();
  }

  // Gestiona la liberación de botones.
  public void mouseReleased(MouseEvent me) {
    // Guarda las coordenadas
    mouseX = me.getX();
    mouseY = me.getY();
    msg = "Arriba";
    repaint();
  }

  // Gestiona el arrastre del ratón
  public void mouseDragged(MouseEvent me) {
    // Guarda las coordenadas
    mouseX = me.getX();
    mouseY = me.getY();
    msg = "*";
    showStatus("Arrastrando el ratón en " + mouseX + ", " + mouseY);
    repaint();
  }

  // Gestiona los movimientos del ratón.
  public void mouseMoved(MouseEvent me) {
    // Muestra el estatus
    showStatus("Moviendo el ratón en " + me.getX () + ", " + me.getY());
  }

  // Muestra msg en la ventana del applet en la actual posición
  public void paint(Graphics g) {
    g.drawString(msg, mouseX, mouseY);
  }
}
```

Una muestra de la salida del programa es la siguiente:



Vamos a ver más detalladamente este ejemplo. La clase **MouseEvents** extiende a **Applet** e implementa a las interfaces **MouseListener** y **MouseMotionListener**. Estas dos interfaces tienen métodos que reciben y procesan los distintos tipos de eventos de ratón. Hay que tener en cuenta que el applet es tanto la fuente como el listener para estos eventos. Esto funciona porque **Component**, que suministra los métodos **addMouseListener( )** y **addMouseMotionListener( )**, es una

superclase de **Applet**. Ser tanto la fuente como el listener para los eventos es algo normal en los applets.

Dentro del método **init( )**, el applet se declara a sí mismo como listener para eventos de ratón. Esto se hace utilizando **addMouseListener( )** y **addMouseMotionListener( )**, que, como ya se ha dicho, son miembros de **Component**. Se muestran a continuación:

void addMouseListener(MouseListener *ml*)
void addMouseMotionListener(MouseMotionListener *mml*)

Donde *ml* es una referencia al objeto que recibe los eventos de ratón, y *mml* es una referencia al objeto que recibe los eventos de movimiento del ratón. En este programa se utiliza el mismo objeto para las dos cosas.

De esta forma, el applet implementa todos los métodos definidos mediante las interfaces **MouseListener** y **MouseMotionListener**. Éstas son las que gestionan los diferentes eventos de ratón. Cada método gestiona su evento y luego devuelve el control.

La gestión de eventos de teclado

Para gestionar eventos de teclado se utiliza la misma arquitectura que se acaba de ver para los eventos de ratón en la sección anterior. La diferencia es, obviamente, que se implementará la interfaz **KeyListener**.

Antes de ver un ejemplo, repasemos cómo se generan los eventos de teclado. Cuando se presiona una tecla, se genera un evento **KEY_PRESSED**, y se llama al método manejador de eventos **keyPressed( )**. Cuando se libera una tecla, se genera un evento **KEY_RELEASED** y se ejecuta el manejador **keyReleased( )**. Si se genera un carácter, entonces se envía un evento **KEY_TYPED** y se invoca al manejador **keyTyped( )**. Es decir, cada vez que el usuario presiona una tecla, se generan al menos dos eventos, y muchas veces tres. Si al programador sólo le interesan los caracteres reales, es posible olvidar la información que se pasa a través de los eventos de presionar y liberar una tecla. Pero si el programa necesita gestionar teclas especiales, como son las teclas de dirección (flechas) o las teclas de función, entonces hay que echar un vistazo a esa información con el manejador **keyPressed( ).**

En el siguiente programa se puede ver una entrada por teclado. El ejemplo envía las teclas pulsadas a la ventana del applet y muestra el estado "presionada/liberada" de cada tecla en la ventana de estado.

```
// Ejemplo de los manejadores de eventos de teclado.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="SimpleKey" width=300 height=l00>
  </applet>
*/

public class SimpleKey extends Applet
  implements KeyListener {

  String msg = "";
  int X = 10, y = 20; // Salida de las coordenadas

  public void init() {
    addKeyListener(this);
  }
```

```
  public void keyPressed(KeyEvent ke) {
    showStatus("Tecla Abajo");
  }

  public void keyReleased(KeyEvent ke) {
    showStatus("Tecla Arriba");
  }

  public void keyTyped(KeyEvent ke) {
    msg += ke.getKeyChar();
    repaint();
  }

  / / Muestra la pulsación de las teclas.
  public void paint(Graphics g) {
    g.drawString(msg, X, Y);
  }
}
```

Un ejemplo de la salida del programa es la siguiente:

Si se quieren gestionar las teclas especiales, como las teclas de dirección (flechas) o las teclas de función, se debe utilizar el manejador **keyPressed( )** para responder a ellas, ya que no están disponibles a través de **keyTyped( )**. Para identificar las teclas, hay que utilizar sus códigos de tecla virtuales. Por ejemplo, con el siguiente applet se muestra el nombre de algunas teclas especiales:

```
// Ejemplo de algunos códigos de tecla virtuales.
import java.awt.*;
import java.awt.event .*;
import java.applet.*;
/*
<applet code="KeyEvents" width=300 height=l00>
</applet>
*/

public class KeyEvents extends Applet
  implements KeyListener {

  String msg = "";
  int X = 10, Y = 20; // coordenadas de la salida

  public void init() {
    addKeyListener(this);
  }

  public void keyPressed(KeyEvent ke) {
  showStatus("Tecla Abajo");
```

```
  int key = ke.getKeyCode();
  switch(key) {
    case KeyEvent.VK_Fl:
      msg += "<Fl>";
      break;
    case KeyEvent.VK_F2:
      msg += "<F2>";
      break;
    case KeyEvent.VK_F3:
      msg += "<F3>";
      break;
    case KeyEvent.VK_PAGE_DOWN:
      msg += "<PgDn>";
      break;
    case KeyEvent.VK_PAGE_UP:
      msg += "<PgUp>" ;
      break;
    case KeyEvent.VK_LEFT:
      msg += "<Flecha Izquierda>" ;
      break;
    case KeyEvent.VK_RIGHT:
      msg += "<Flecha Derecha>" ;
      break;
    }
    repaint();
  }
  public void keyReleased(KeyEvent ke) {
    showStatus("Tecla Arriba");
  }
  public void keyTyped(KeyEvent ke) {
    msg += ke.getKeyChar();
    repaint();
  }
  // Muestra la pulsación de teclas.
  public void paint(Graphics g) {
    g.drawString(msg, X, Y};
  }
}
```

Un ejemplo de la salida se muestra a continuación:

Los procedimientos que se han mostrado en los ejemplos anteriores de eventos de ratón y teclado se pueden generalizar a cualquier tipo de gestión de eventos, incluidos aquellos eventos

generados por controles. En capítulos posteriores, se verán muchos ejemplos que gestionan otro tipo de eventos, pero todos ellos seguirán la misma estructura básica que los programas que se acaban de mostrar.

TABLA 22-4
Interfaces de listener, comúnmente utilizadas, implementadas por las clases Adapter

Clase adaptadora	Interfaz para listener
ComponentAdapter	ComponentListener
ContainerAdapter	ContainerListener
FocusAdapter	FocusListener
KeyAdapter	KeyListener
MouseAdapter	MouseListener
MouseMotionAdapter	MouseMotionListener
Window Adapter	WindowListener

Clases adaptadoras

Java proporciona una característica especial, llamada *clase adaptadora*, que en algunas circunstancias puede simplificar la creación de manejadores de eventos. Una clase adaptadora proporciona una implementación vacía de todos los métodos en una interfaz listener de evento. Estas clases adaptadoras son útiles cuando se quiere recibir y procesar sólo alguno de los eventos que está gestionando una interfaz listener de eventos en particular. Se puede definir una nueva clase para actuar como un listener de eventos extendiendo una de las clases adaptadoras e implementando sólo aquellos eventos que interesen.

Por ejemplo, la clase **MouseMotionAdapter** tiene dos métodos, **mouseDragged( )** y **mouseMoved( )** los cuales son métodos definidos por la interfaz **MouseMotionListener**. Si sólo estamos interesados en eventos de arrastre de ratón, se puede simplemente extender **MouseMotionAdapter** y sobrescribir **mouseDragged( )**. La implementación vacía de **mouseMoved( )** gestionaría los eventos de movimiento de ratón por nosotros.

En la Tabla 22-4 se enlistan las diferentes clases adaptadoras comúnmente utilizadas en **java.awt.event** y la interfaz que implementa cada una.

El siguiente ejemplo es una muestra de una clase adaptadora. El ejemplo visualiza un mensaje en la barra de estado de un visor de applets o de un navegador cuando se arrastra o se hace clic con el ratón. Sin embargo, todos los demás eventos del ratón son silenciosamente ignorados. El programa tiene tres clases. **AdapterDemo** extiende a **Applet**. Su método **init( )** crea una instancia de **MyMouseAdapter** y declara al objeto para que pueda recibir notificaciones de eventos de ratón. También crea una instancia de **MyMouseMotionAdapter** y declara al objeto para que pueda recibir notificaciones de eventos de movimiento de ratón. Los dos constructores reciben una referencia al applet como argumento.

MyMouseAdapter extiende de **MouseAdapter** y sobrescribe el método **mouseClicked( )**. Los otros eventos de ratón se ignoran mediante código heredado de la clase **MouseAdapter**. **MyMouseMotionAdapter** extiende de **MouseMotionAdapter** y sobrescribe el método **mouseDragged( )**. Los otros eventos de movimiento de ratón se ignoran mediante código heredado de la clase **MouseMotionAdapter**.

Es importante notar que las dos clases de listener de eventos guardan una referencia al applet. Esta información se proporciona a sus constructores como un argumento y se utiliza posteriormente para invocar al método **showStatus( )**.

```
// Ejemplo de un adaptador.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="AdapterDemo" width=300 height=l00>
  </applet>
*/
public class AdapterDemo extends Applet {
  public void init() {
    addMouseListener(new MyMouseAdapter(this));
    addMouseMotionListener(new MyMouseMotionAdapter(this));
  }
}
class MyMouseAdapter extends MouseAdapter {
  AdapterDemo adapterDemo;
  public MyMouseAdapter(AdapterDemo adapterDemo) {
    this.adapterDemo = adapterDemo;
  }
  // Gestiona los clics del ratón.
  public void mouseClicked(MouseEvent me) {
    adapterDemo.showStatus("El ratón tuvo un clic");
  }
}
class MyMouseMotionAdapter extends MouseMotionAdapter {
  AdapterDemo adapterDemo;
  public MyMouseMotionAdapter(AdapterDemo adapterDemo) {
    this.adapterDemo = adapterDemo;
  }
  // Gestiona el arrastre del ratón.
  public void mouseDragged(MouseEvent me) {
    adapterDemo.showStatus("Ratón arrastrado");
  }
}
```

Como se puede ver en el programa, no se tienen que implementar todos los métodos definidos por las interfaces **MouseMotionListener** y **MouseListener**, lo que ahorra un considerable esfuerzo y previene contra posibles confusiones con los métodos vacíos. Como ejercicio, se podría intentar reescribir uno de los ejemplos de entrada por teclado mostrados anteriormente para hacer que utilice **KeyAdapter**.

Clases internas

En el Capítulo 7 se explicaron los fundamentos básicos de las clases internas. Ahora se verá por qué son importantes. Recuerde que una *clase interna* es una clase definida dentro de otra clase, o incluso dentro de una expresión. Este apartado muestra cómo se pueden utilizar las clases internas para simplificar el código cuando se utilizan clases adaptadoras de eventos.

Para comprender las ventajas de las clases internas, consideremos el applet mostrado a continuación. Este *no* utiliza una clase interna. Su objetivo es visualizar la cadena "Se ha

presionado el ratón" en la barra de estado del visor de applets o del navegador cuando se presione un botón del ratón.

En este programa hay dos clases de alto nivel. **MousePressedDemo** que extiende a **Applet**, y **MyMouseAdapter** que extiende a **MouseAdapter**. El método **init( )** de **MousePressedDemo** instancia a **MyMouseAdapter** y pasa ese objeto como un argumento al método **addMouseListener( )**.

Es importante observar que se pasa una referencia al applet como argumento al constructor **MyMouseAdapter**. Esta referencia se guarda en una variable de instancia para que la utilice más tarde el método **mousePressed( )**. Cuando se presiona el botón del ratón, se invoca al método **showStatus( )** del applet a través de la referencia al applet que se tenía guardada. En otras palabras, se invoca a **showStatus( )** en función de la referencia guardada por **MyMouseAdapter**.


```
// Este applet NO utiliza una clase interna.
import java.applet.*;
import java.awt.event.*;
/*
  <applet code="MousePressedDemo" width=200 height=100>
  </applet>
*/

public class MousePressedDemo extends Applet {
  public void init() {
    addMouseListener(new MyMouseAdapter(this));
  }
}

class MyMouseAdapter extends MouseAdapter {
  MousePressedDemo mousepressedDemo;
  public MyMouseAdapter(MousePressedDemo mousePressedDemo) {
    this.mousePressedDemo = mousepressedDemo;
  }
  public void mousePressed(MouseEvent me) {
    mousePressedDemo.showStatus("Se ha presionado el ratón.");
  }
}
```

A continuación se muestra cómo se puede mejorar el programa anterior utilizando una clase interna. Aquí, **InnerClassDemo** es una clase de nivel superior que extiende a **Applet**. **MyMouseAdapter** es una clase interna que extiende a **MouseAdapter**. Puesto que se define a **MyMouseAdapter** dentro del ámbito de **InnerClassDemo**, tiene acceso a todos los métodos y variables dentro del ámbito de esa clase. Por lo tanto, el método **mousePressed( )** puede llamar directamente al método **showStatus( )**. Al hacer esto, ya no se necesita guardar referencia alguna al applet. Es decir, ya no es necesario que **MyMouseAdapter( )** pase una referencia al objeto invocado.

```
// Ejemplo de una clase interna.
import java.applet.*;
import java.awt.event.*;
/*
  <applet code="InnerClassDemo" width=200 height=100>
  </applet>
*/
```

```
public class InnerClassDemo extends Applet {
  public void init() {
    addMouseListener(new MyMouseAdapter());
  }
  class MyMouseAdapter extends MouseAdapter {
    public void mousePressed(MouseEvent me) {
      showStatus("Se ha presionado el ratón");
    }
  }
}
```

Clases internas anónimas

Una clase interna *anónima* es aquella a la que no se le asigna un nombre. En este apartado se muestra cómo es más fácil la escritura de manejadores de eventos utilizando clases internas anónimas. Considere al applet que se muestra a continuación. Su objetivo es, al igual que antes, visualizar la cadena "Se ha presionado el ratón" en la barra de estado del visor de applets o del navegador cuando se presione el ratón.

```
// Ejemplo de una clase interna anónima.
import java.applet.*;
import java.awt.event.*;
/*
  <applet code="AnonymousInnerClassDemo" width=200 height=l00>
  </applet>
*/
public class AnonymousInnerClassDemo extends Applet {
  public void init() {
    addMouseListener(new MouseAdapter() {
      public void mousePressed(MouseEvent me) {
        showStatus("Se ha presionado el ratón");
      }
    });
  }
}
```

En este programa hay una clase de alto nivel: **AnonymousInnerClassDemo**. El método **init( )** llama al método **addMouseListener( )**. Su argumento es una expresión que define y cita una clase interna anónima. Es interesante analizar con detalle esta expresión.

La sintaxis **new MouseAdapter( ) { ... }** indica al compilador que el código que está entre llaves define una clase interna anónima. Además, esa clase extiende a **MouseAdapter**. Esta nueva clase no tiene nombre, pero es automáticamente solicitada cuando se ejecuta esta expresión.

Debido a que se define esta clase interna anónima dentro del ámbito de **AnonymousInnerClassDemo**, ésta tiene acceso a todos los métodos y variables dentro del ámbito de esa clase. Por lo tanto, puede llamar directamente al método **showStatus( )**.

Como se acaba de mostrar, tanto las clases internas anónimas como las no anónimas resuelven algunos molestos problemas de una manera efectiva y sencilla. También permiten crear un código más eficiente.

23 CAPÍTULO

AWT: trabajando con ventanas, gráficos y texto

En el Capítulo 21, se dio una introducción al conjunto de herramientas gráficas AWT (Abstract Window Toolkit) utilizándolo en varios applets. En este capítulo se realizará un estudio más detallado. El AWT contiene numerosos métodos y clases que permiten crear y gestionar ventanas. AWT es también la base sobre la cual Swing está construido. AWT es grande y una descripción completa podría llenar un libro completo, por lo que no se describirán con detalle todos los métodos, variables de instancia o clase del AWT. Sin embargo, sí se explicarán, en este capítulo y en los dos siguientes, todas las técnicas necesarias para utilizar eficazmente el AWT cuando se crean applets o programas independientes. Partiendo de esta información, le resultará fácil explorar otras partes del AWT y luego aprender Swing.

En este capítulo aprenderemos a crear y gestionar ventanas, administrar tipos de letra, texto de salida, y utilizar gráficos. En el Capítulo 24 se describen los diferentes controles del AWT, como las barras de desplazamiento y los botones. También se explicarán otros aspectos del mecanismo de gestión de eventos de Java. En el Capítulo 25 se examinará la animación y el subsistema de imágenes del AWT.

Aunque un uso común del AWT es dar el soporte necesario a las ventanas de los applets, también se puede utilizar para crear ventanas independientes que se ejecuten en un entorno gráfico de interfaz de usuario, como Windows. La mayoría de los ejemplos que se presentan en este capítulo son applets, por lo que para ejecutarlos, es necesario utilizar un visor de applets o un navegador compatible con Java. También hay unos cuantos ejemplos que muestran cómo crear programas independientes con ventanas.

Antes de comenzar es importante mencionar que hoy en día muchos programadores que utilizan Java prefieren programar interfaces graficas basadas en Swing. Esto debido a que Swing proporciona una implementación más rica, que AWT, de algunos controles comunes, como botones y listas. Es fácil creer que AWT, por tanto, ya no es importante y que ha sido sustituido por Swing. Sin embargo, esta suposición no es del todo correcta. Como se mencionó, Swing está construido sobre AWT. Así que muchas características de AWT son también características de Swing. Además muchas clases de AWT son usadas directa o indirectamente por Swing. Finalmente, para algunos tipos de programas pequeños (especialmente applets pequeños) que sólo requieren una interfaz gráfica sencilla, usar AWT es mejor que utilizar Swing. Por ende, aunque hoy en día muchas interfaces estén basadas en Swing, un conocimiento sólido de AWT es aún necesario. Dicho de forma simple, no es posible ser un buen programador de Java sin conocer AWT.

NOTA *Si todavía no ha leído el Capítulo 22, por favor hágalo ahora. Ese capítulo proporciona una visión general de la gestión de eventos, la cual se utiliza en varios de los ejemplos de este capítulo.*

Las clases de AWT

El paquete **java.awt** contiene todas las clases del AWT. Éste es uno de los paquetes más grandes de Java. Afortunadamente, se puede comprender y manejar con relativa facilidad, ya que dicho paquete está organizado de manera jerárquica. La Tabla 23-1 muestra algunas de las clases AWT.

Clase	Descripción
AWTEvent	Encapsula eventos del AWT.
AWTEventMulticaster	Envía eventos a varios auditores (listeners).
BorderLayout	Es un gestor de organización. Utiliza cinco componentes: North (Norte), South (Sur), East (Este), West (Oeste), y Center (centro).
Button	Crea un botón (push button).
Canvas	Crea una ventana en blanco sin una semántica asociada.
CardLayout	Es un gestor de organización. Simula tarjetas indexadas; sólo se enseña la que está en la parte superior.
Checkbox	Crea un control de tipo checkbox.
CheckboxGroup	Crea un grupo de controles checkbox.
CheckboxMenuItem	Crea un elemento de menú de tipo on /off.
Choice	Crea una lista emergente.
Color	Gestiona los colores en un entorno portátil independiente de la plataforma.
Component	Es una superclase abstracta de varios componentes del AWT.
Container	Es una subclase de **Component** que puede tener otros componentes.
Cursor	Encapsula un cursor definido como un mapa de bits.
Dialog	Crea una ventana de diálogo.
Dimension	Especifica las dimensiones de un objeto. El ancho se almacena en **width**, y el alto en **height**.
Event	Encapsula eventos.
EventQueue	Pone eventos en cola.
FileDialog	Crea una ventana desde la que se puede seleccionar un archivo.
FlowLayout	Es un gestor de organización que sitúa los componentes de izquierda a derecha y de arriba abajo.
Font	Encapsula un tipo de letra.
FontMetrics	Encapsula información relacionada con un tipo de letra. Esta información ayuda a mostrar texto en una ventana.
Frame	Crea una ventana estándar que tiene una barra de título, esquinas que permiten cambiar el tamaño de la ventana, y una barra de menú.

TABLA 23-1 Algunas clases del AWT

Clase	Descripción
Graphics	Encapsula el contexto gráfico. Este contexto lo utilizan varios métodos para mostrar información en una ventana.
GraphicsDevice	Describe un dispositivo gráfico, como puede ser una pantalla o una impresora.
GraphicsEnvironment	Describe la colección de objetos **Font** y **GraphicsDevice** disponibles.
GridBagConstraints	Define varias restricciones relacionadas con la clase **GridBagLayout**.
GridBagLayout	Es un gestor de organización en cuadrícula. Muestra los componentes con las restricciones especificadas por **GridBagConstraint**.
GridLayout	Es un gestor de organización en cuadrícula. Muestra los componentes en una cuadrícula de dos dimensiones.
Image	Encapsula imágenes gráficas.
Insets	Encapsula los bordes de un contenedor.
Label	Crea una etiqueta que muestra un texto.
List	Crea una lista de donde el usuario puede elegir un elemento. Es similar a la lista estándar de Windows.
MediaTracker	Gestiona objetos multimedia.
Menu	Crea un menú desplegable.
MenuBar	Crea una barra de menú.
MenuComponent	Clase abstracta implementada por varias clases relacionadas con los menús.
MenuItem	Crea un elemento de un menú.
MenuShortcut	Encapsula un atajo vía teclado para un elemento de menú.
Panel	Es la subclase concreta más simple de **Container**.
Point	Encapsula las coordenadas cartesianas almacenadas en **x**, **y**.
Polygon	Encapsula un polígono.
PopupMenu	Encapsula un menú emergente (pop-up).
PrintJob	Clase abstracta que representa una tarea de impresión.
Rectangle	Encapsula un rectángulo.
Robot	Soporta la verificación automática para aplicaciones basadas AWT.
Scrollbar	Crea un control barra de desplazamiento
ScrollPane	Contenedor que proporciona barras de desplazamiento horizontal y/o vertical para otro componente.
SystemColor	Contiene los colores de los elementos gráficos como ventanas, barras de desplazamiento, texto, etc.
TextArea	Crea el control para un área de texto con varias líneas.
TextComponent	Es una superclase de **TextArea** y **TextField**.
TextField	Crea el control para un área de texto con una sola línea.
Toolkit	Clase abstracta implementada por el AWT.
Window	Crea una ventana sin marco, sin menú y sin título

TABLA 23-1 Algunas clases del AW

Aunque la estructura básica de AWT ha sido la misma desde la versión 1.0 de Java, algunos de los métodos originales se han quedado obsoletos y han sido reemplazados por otros nuevos. Para mantener la compatibilidad hacia atrás Java aún soporta todos los métodos de la versión 1.0, aunque no se describen en este libro porque no se recomienda su uso en los programas actuales.

Fundamentos básicos de ventanas

El AWT define ventanas de acuerdo con una jerarquía de clases que da funcionalidad y carácter específico a cada nivel. Los dos tipos de ventanas más utilizadas son las que derivan de **Panel**, que son empleadas por los applets, y las que derivan de **Frame** que permiten crear aplicaciones estándar con ventanas. La mayor parte de la funcionalidad de estas ventanas se hereda de sus superclases. Por eso, para comprenderlas es importante hacer una descripción de la jerarquía de clases de ambas clases. En la Figura 23-1 se muestra la jerarquía de clases de **Panel** y **Frame**. Veamos ahora cada una de estas clases.

Component

En la parte superior de la jerarquía de AWT se encuentra la clase **Component. Component** es una clase abstracta que encapsula todos los atributos de un componente visual. Todos los elementos de la interfaz de usuario que se visualizan en la pantalla y que interactúan con el usuario son subclases de **Component**. Esta clase define unos cien métodos públicos que son responsables de la gestión de eventos, como la entrada por teclado o ratón, cambio de tamaño y posición de la ventana, y del repintado de una ventana. En los Capítulos 21 y 22 ya se han utilizado muchos de estos métodos al crear applets. Un objeto **Component** es el responsable de recordar los colores de fondo y de frente así como el tipo de letra en uso.

Container

La clase Container es una subclase de **Component**. Contiene métodos adicionales que permiten que otros objetos de la clase **Component** puedan estar contenidos o ser anidados en un objeto de esta clase. Se pueden almacenar objetos de la clase **Container** dentro de otro objeto de la clase **Container** (dado que ellos mismos son instancias de la clase **Component**). Esto permite generar un sistema de contenidos jerárquico. Un contenedor es el responsable de colocar y situar

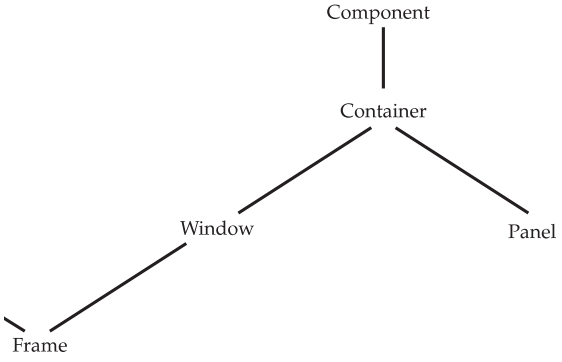


FIGURA 23-1 La jerarquía de clases para **Panel** y **Frame**

los componentes que contenga. Para ello utiliza varios gestores de organización, que se verán en el Capítulo 24.

Panel

La clase **Panel** es una subclase concreta de **Container**. No añade ningún método nuevo, simplemente implementa a **Container**. Se puede decir que un objeto de la clase **Panel** es un componente de pantalla concreto, anidable recursivamente. La clase **Panel** es la superclase de la clase **Applet**. Cuando la salida por pantalla se dirige a un applet, se dibuja en la superficie de un objeto de la clase **Panel**. Básicamente, un objeto de la clase **Panel** es una ventana que no contiene ni barra de título, ni barra de menú, ni bordes. Por eso no se pueden ver dichos elementos cuando se ejecuta un applet con un navegador, pero sí cuando se hace con un visor de applets, ya que el visor proporciona el título y los bordes.

Se pueden añadir componentes a un objeto de la clase **Panel** utilizando el método **add( )** (heredado de la clase **Container**). Una vez que se han añadido componentes, se puede colocar y cambiar su tamaño utilizando los métodos **setLocation( ), setSize( ), setPreferredSize( ),** o **setBounds( )** definidos por la clase **Component**.

Window

La clase **Window** crea una ventana de nivel superior. Una ventana de nivel superior no está contenida en ningún objeto, sino que se encuentra situada directamente sobre el escritorio. No es habitual crear directamente objetos de la clase **Window**. Generalmente se crean objetos de la clase **Frame**, que es una subclase de la clase **Window. Frame** se describe a continuación.

Frame

La clase **Frame** encapsula lo que normalmente se conoce como"ventana". Es una subclase de **Window** y tiene una barra de título, una barra de menú, bordes y esquinas para cambiar el tamaño. Si se crea un objeto de la clase **Frame** desde un applet, aparecerá un mensaje de aviso (con el texto"Java Applet Window") indicando que se ha creado una ventana applet. Este mensaje avisa al usuario que la ventana que está viendo ha sido arrancada por un applet y no por un programa. Un applet que pudiese hacerse pasar por una aplicación local podría ser utilizado para obtener claves, u otro de tipo de información, sin que el usuario se diese cuenta. Cuando se crea una ventana de tipo **Frame** mediante un programa en lugar de con un applet, la ventana tiene una apariencia normal.

Canvas

Aunque no forma parte de la jerarquía de ventanas para applets o aplicaciones, existe otro tipo de ventana que es interesante conocer: **Canvas. Canvas** encapsula una ventana vacía sobre la que se puede dibujar. Más adelante en este libro veremos un ejemplo de **Canvas**.

Trabajo con ventanas de tipo Frame

Después del applet, el tipo de ventana que más comúnmente se emplea es el derivado de la clase **Frame**. Se utilizan para crear ventanas hijas dentro de los applets, y ventanas de nivel superior o hijas en aplicaciones. Como ya se ha comentado anteriormente, Frame crea una ventana de estilo estándar.

A continuación se muestran dos constructores de la clase **Frame**:

Frame( )
Frame(String *nombre*)

El primero crea una ventana estándar que no tiene título. El segundo crea una ventana con el título especificado en la cadena nombre. Observe que las dimensiones de la ventana no se especifican aquí. Las dimensiones de la ventana se establecen después de que se ha creado la ventana.

Vamos a ver a continuación algunos de los muchos métodos que se utilizarán al trabajar con ventanas de tipo **Frame**.

Cómo establecer las dimensiones de una ventana

El método **setSize( )** se utiliza para establecer las dimensiones de una ventana. Su firma es la siguiente:

void setSize(int *newWidth*, int *newHeight*)
void setSize(Dimension *newSize*)

El nuevo tamaño de la ventana se especifica con los valores *newWidth* y *newHeight*, o con los campos **width** y **height** del objeto **Dimension** pasado como newSize. Las dimensiones se dan en pixeles.

El método **getSize( )** se utiliza para obtener el tamaño actual de una ventana. Su firma es la siguiente:

Dimension getSize( )

Este método devuelve el tamaño actual de la ventana; la información se devuelve en los campos **width** y **height** de un objeto **Dimension**.

Ocultar y mostrar una ventana

Después de crear una ventana de tipo frame, no será visible hasta que no se llame al método **setVisible( )**, que tiene la siguiente firma:

void setVisible(boolean *visibleFlag*)

El componente es visible si el argumento visibleFlag es **true**. En caso contrario, permanecerá oculto.

Poner el título a una ventana

Se puede cambiar el título a una ventana de tipo frame con el método **setTitle( )**, que tiene la siguiente forma:

void setTitle(String *newTitle*)

Aquí, newTitle es el nuevo título de la ventana.

Cerrar una ventana de tipo frame

Cuando se utilice una ventana de tipo Frame, el programa debe hacerse cargo de eliminarla de la pantalla cuando se indique a la ventana cerrarse. Esto se logra llamando al método **setVisible(false).** Para saber cuando el usuario solicita el cierre la ventana, se debe implementar el método **windowClosing( )** de la interfaz **WindowListener**. En el método **windowClosing( )** se debe borrar la ventana de la pantalla. Esta técnica se muestra en el ejemplo de la siguiente sección.

Crear una ventana de tipo frame en un Applet

Aunque es posible dentro de un applet crear una ventana creando un objeto o instancia de la clase **Frame**, casi nunca se hace, ya que no es posible hacer mucho con ella. Por ejemplo, no se podrán recibir ni procesar eventos que se produzcan dentro de ella, o mostrar información. La mayoría de las veces se suele crear una subclase de **Frame**, con la que se pueden sobrescribir los métodos de **Frame** y gestionar eventos.

Crear una nueva ventana de tipo frame en un applet es bastante fácil. Primero, se crea una subclase de **Frame**. Después, se sobrescribe cualquiera de los métodos estándar del applet, como **init( ), start( )**, o **stop( )** para mostrar u ocultar el frame, según se requiera. Por último, se implementa el método **windowClosing( )** de la interfaz **WindowListener**, llamando a **setVisible(false)** cuando se desee cerrar la ventana.

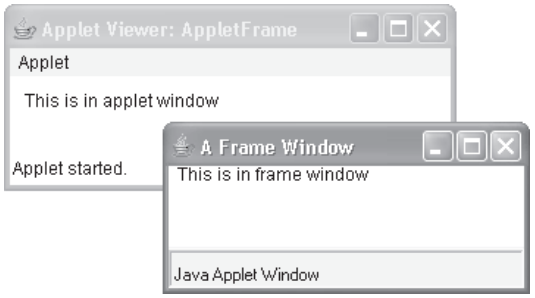
Una vez que se ha definido una subclase **Frame**, se puede crear un objeto de esa clase. Esto hace que se cree una ventana, aunque inicialmente no sea visible. Se le puede hacer visible llamando a **setVisible( )**. Cuando se crea la ventana, se le asigna un alto y un ancho por omisión. Luego se puede establecer el tamaño de la ventana de forma explícita llamando al método **setSize( )**.

El siguiente applet crea un objeto derivado de la clase **Frame**, llamado **EjemploFrame**. En el método **init( )** de la clase **AppletFrame** se crea una ventana de la clase **EjemploFrame**. Observe que **EjemploFrame** llama al constructor de **Frame**. Esto hace que se cree una ventana frame estándar con el título que se indica en la variable **titulo**. Este ejemplo sobrescribe los métodos **start( )** y **stop( )** de la ventana applet para que muestren y oculten la ventana hija, respectivamente. Esto hace que la ventana se borre automáticamente cuando se acaba el applet, cuando se cierra la ventana o, si se utiliza un navegador, cuando uno se cambia a otra página. También hace que se muestre la ventana hija cuando el navegador vuelve a mostrar al applet.

```
/ / Crear una ventana tipo Frame hija dentro de un applet.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="AppletFrame" width=300 height=50>
  </applet>
*/

// Crea una subclase de Frame.
class EjemploFrame extends Frame {
  EjemploFrame(String titulo) {
    super(titulo);
    // crea un objeto para tratar los eventos de la ventana
    MiVentanaAuditora unAuditor = new MiVentanaAuditora(this);
    // lo registra para recibir dichos eventos
    addWindowListener(unAuditor);
  }
  public void paint(Graphics g) {
    g.drawString("Esto es una ventana de tipo Frame" , 10, 40);
  }
}

class MiVentanaAuditora extends WindowAdapter {
  EjemploFrame ejemploFrame;
  public MiVentanaAuditora(EjemploFrame ejemploFrame) {
```

```
    this.ejemploFrame = ejemploFrame;
  }
  public void windowClosing(WindowEvent we) {
    ejemploFrame.setVisible(false);
  }
}

// Crea una ventana de tipo Frame
public class AppletFrame extends Applet {
  Frame f;
  public void init() {
    f = new EjemploFrame("Una ventana Frame");

    f.setSize(250, 250);
    f.setVisible(true);
  }
  public void start() {
    f.setVisible(true);
  }
  public void stop() {
    f.setVisible(false);
  }
  public void paint(Graphics g) {
    g.drawString("Esto esta en la ventana del applet", 10, 20);
  }
}
```

La salida de este programa es la siguiente:



Gestión de eventos en una ventana de tipo Frame

Como **Frame** es una subclase de **Component**, hereda todas las capacidades definidas por **Component**. Esto quiere decir que se puede utilizar y gestionar una ventana de tipo Frame de la misma forma que se gestiona la ventana principal de un applet. Por ejemplo, se puede sobrescribir **paint( )** para visualizar algo en la ventana, llamar a **repaint( )** cuando se necesite restaurar la ventana, y añadir manejo de eventos. En cualquier momento que se produzca un evento en una ventana, se llama a los métodos responsables del manejo de eventos definidos por esa ventana. Cada ventana gestiona sus propios eventos. Por ejemplo, el siguiente programa crea una ventana que responde a los eventos de ratón. La ventana principal del applet también responde a los eventos de ratón. Cuando se ejecute este programa, se podrá ver que se envían los eventos de ratón a la ventana en que ha ocurrido el evento.

```
// Gestión de eventos de ratón tanto en la ventana hija como en la ventana applet.
import java.awt.*;
```

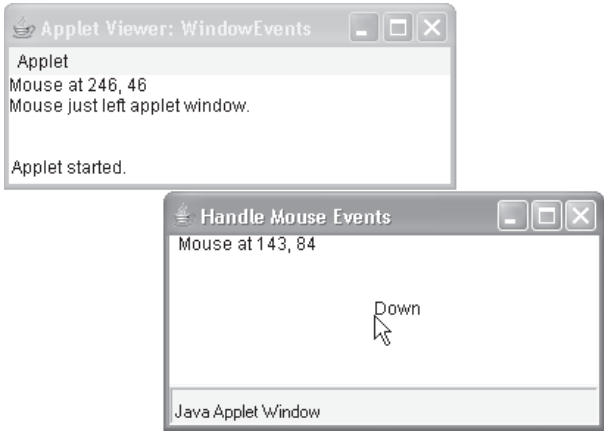
```
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="VentanaEventos" width=300 height=50>
  </applet>
*/

// Crea una subclase de Frame.
class EjemploFrame extends Frame
implements MouseListener, MouseMotionListener {

String msg = "";
int ratonX=10, ratonY=40;
int movX=0, movY=0;

EjemploFrame(String titulo) {
  super(titulo);
  // registra este objeto para recibir sus propios eventos de ratón
  addMouseListener(this);
  addMouseMotionListener(this);
  // crea un objeto para manejar los eventos de la ventana
  MiVentanaAuditora unAuditor = new MiVentanaAuditora (this);
  // lo registra para recibir esos eventos
  addWindowListener(unAuditor);
}

// Gestiona el evento mouseClicked, que produce el botón del ratón al ser
pulsado
public void mouseClicked(MouseEvent yo) {
}

// Gestiona el evento mouseEntered, que se produce si el apuntador del ratón
entra en la ventana
public void mouseEntered(MouseEvent evtObj) {
  / / guarda las coordenadas
  ratonX = 10;
  ratonY = 54;
  msg = "El ratón acaba de entrar en la ventana hija.";
  repaint();
}

// Gestiona el evento mouseExited que se produce si el apuntador del ratón sale
de la ventana
public void mouseExited(MouseEvent evtObj) {
  // guarda las coordenadas
  ratonX = 10;
  ratonY = 54;
  msg = "El ratón acaba de salir de la ventana hija.";
  repaint();
}

// Gestiona el evento mousePressed, que se produce si se presiona el botón del
ratón.
public void mousePressed(MouseEvent yo) {
  // guarda las coordenadas
  ratonX = yo.getX();
  ratonY = yo.getY();
  msg = "Abajo";
  repaint();
}
```

```
  // Gestiona el evento mouseReleased, que se produce si se libera el botón del
  ratón.
  public void mouseReleased(MouseEvent yo) {
    // guarda las coordenadas
    ratonX = yo.getX();
    ratonY = yo.getY();
    msg = "Arriba";
    repaint();
}
  // Gestiona el evento mouseDragged, que se produce si se arrastra el ratón
  con  el botón presionado
  public void mouseDragged(MouseEvent yo) {
    // guarda las coordenadas
    ratonX = yo.getX();
    ratonY = yo.getY();
    movX = yo.getX();
    movY = yo.getY();
    msg = "*";
    repaint() ;
  }
  // Gestiona el evento mouseMoved, que se produce si se mueve el ratón.
  public void mouseMoved(MouseEvent yo) {
  // guarda las coordenadas
    movX = yo.getX();
    movY = yo.getY();
    repaint(0,0, 100, 60);
  }
  public void paint(Graphics g) {
    g.drawString(msg, ratonX, ratonY);
    g.drawString("El ratón está en la posición" + movX + ", " + movY, 10, 40);
  }
}
class MiVentanaAuditora extends WindowAdapter {
  EjemploFrame ejemploFrame;
  public MiVentanaAuditora (EjemploFrame ejemploFrame) {
  this.ejemploFrame = ejemploFrame;
  }
  public void windowClosing(WindowEvent we) {
  ejemploFrame.setVisible(false);
  }
}
// Ventana del applet
public class VentanaEventos extends Applet
  implements MouseListener, MouseMotionListener {
  EjemploFrame f;
  String msg = "";
  int ratonX=0, ratonY=l0;
  int movX=0,movY=0;
  // Crea una ventana tipo Frame
  public void init() {
    f = new EjemploFrame ("Gestiona los eventos del ratón");
    f.setSize(300, 200);
```

```
  f.setVisible(true);

  // Registra este objeto para que reciba sus propios eventos de ratón.
  addMouseListener(this);
  addMouseMotionListener(this);
}

// Elimina la ventana tipo Frame cuando el applet se detiene
public void stop() {
  f.setVisible(false);
}

// Muestra la ventana tipo Frame cuando comienza el applet.
public void start() {
  f.setVisible(true);
}

// Gestiona el evento mouseClicked.
public void mouseClicked(MouseEvent yo) {
}

// Trata el evento mouseEntered.
public void mouseEntered(MouseEvent yo) {
  // guarda las coordenadas
  ratonX = 0;
  ratonY = 24;
  msg = "El ratón acaba de introducirse en la ventana del applet.";
  repaint();
}

// Gestiona el evento mouseExited.
public void mouseExited(MouseEvent yo) {
  // guarda las coordenadas
  ratonX = 0;
  ratonY = 24;
  msg = "El ratón acaba de salir de la ventana del applet.";
  repaint();
}
// Gestiona el evento mousePressed.
public void mousePressed(MouseEvent yo) {
  // guarda las coordenadas
  ratonX = yo.getX();
  ratonY = yo.getY();
  msg = "Abajo";
  repaint();
}

// Gestiona el evento mouseReleased.
public void mouseReleased(MouseEvent yo) {
  // guarda las coordenadas
  ratonX ="yo.getX();
  ratonY = yo.getY();
  msg = "Arriba";
  repaint();
}

// Gestiona el evento mouseDragged.
public void mouseDragged(MouseEvent yo) {
```

```
    // guarda las coordenadas
    ratonX = yo.getX();
    ratonY = yo.getY();
    movX = yo.getX();
    movY = yo.getY();
    msg = "*";
    repaint();
  }

  // Gestiona el evento mouseMoved.
  public void mouseMoved(MouseEvent yo) {
    // guarda las coordenadas
    movX = yo.getX();
    movY = yo.getY();
    repaint(0,0, 100, 20);
  }

  / / Muestra msg en la ventana del applet.
  public void paint(Graphics g) {
    g.drawString(msg, ratonX, ratonY);
    g.drawString("Ratón en la posición " + movX + ", " + movY, 0, 10);
  }
}
```

La salida del programa es la siguiente:



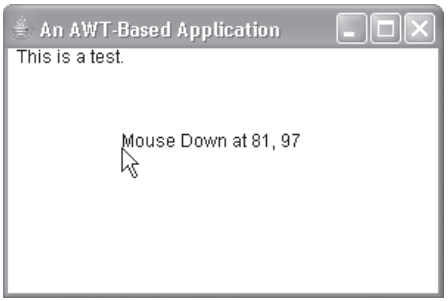
Creación de un programa con ventanas

Generalmente se utiliza AWT de Java para crear applets, pero también se pueden crear aplicaciones independientes basadas en AWT. Para hacer eso, simplemente hay que crear una instancia de la ventana o ventanas que se vayan a necesitar dentro de **main(**). Por ejemplo, el siguiente programa crea una ventana tipo Frame que responde a los clics del ratón y cuando se pulsa una tecla:

```
// Crear una aplicación basada en AWT.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
```

```
// Crea una ventana
public class MiAplicacionVentana extends Frame {
  String keyMensaje = "Esto es una prueba.";
  String mouseMensaje = "";
  int ratonX=30, ratonY=30;

  public MiAplicacionVentana() {
    addKeyListener(new MiKeyAdapter(this));
    addMouseListener(new MiMouseAdapter(this));
    addWindowListener(new MiVentanaAuditora());
  }

  public void paint(Graphics g) {
    g.drawString(keyMensaje, 10, 40);
    g.drawString(mouseMensaje, ratonX, ratonY);
  }

  // Crea la ventana.
  public static void main(String args[]) {
  MiAplicacionVentana unaAplicacionVentana =new MiAplicacionVentana();

    unaAplicacionVentana.setSize(new Dimension(300, 200));
    unaAplicacionVentana.setTitle("Una aplicación basada en el AWT");
    unaAplicacionVentana.setVisible(true);
  }
}

class MiKeyAdapter extends KeyAdapter {
  MiAplicacionVentana appVentana;
  public MiKeyAdapter(MiAplicacionVentana appVentana) {
    this.appVentana = appVentana;
  }
  public void keyTyped(KeyEvent ke) {
    appVentana.keyMensaje += ke.getKeyChar();
    appVentana.repaint();
  };
}

class MiMouseAdapter extends MouseAdapter {
  MiAplicacionVentana appVentana;
  public MiMouseAdapter(MiAplicacionVentana appVentana) {
    this.appVentana = appVentana;
  }
  public void mousePressed(MouseEvent yo) {
    appVentana.ratonX = yo.getX();
    appVentana.ratonY = yo.getY();
    appVentana.mouseMensaje = "Posición del ratón en la posición"+ appVentana.
                              ratonX + ", " + appVentana.ratonY;
    appVentana.repaint();
  }
}

class MiVentanaAuditora extends WindowAdapter {
  public void windowClosing(WindowEvent we) {
    System.exit(0);
  }
}
```

La salida de este programa es la siguiente:



Una vez creada, la ventana tiene vida propia. Observe que **main( )** termina con la llamada a **una Aplicación Ventana.setVisible(true)**. Sin embargo, el programa sigue ejecutándose hasta que se cierre la ventana. Básicamente, cuando se crea una aplicación con ventanas, se utilizará **main( )** para lanzar su ventana de nivel superior. Después, el programa funcionará igual que una aplicación basada en interfaz gráfica y no como los anteriores programas basados en consola.

Visualización de información dentro de una ventana

En el sentido más amplio, una ventana es un contenedor de información. Aunque en los ejemplos anteriores ya hemos mostrado pequeños textos en una ventana, todavía no hemos comenzado a estudiar la capacidad de una ventana para presentar textos de alta calidad y gráficos. Es más, gran parte de la potencia del AWT viene de estos elementos. Por esta razón, en el resto de este capítulo se verán las capacidades de gestión de texto, gráficos y tipos de letra de Java, que son bastante potentes y flexibles.

Trabajo con gráficos

AWT tiene una amplia variedad de métodos gráficos. Todos los gráficos se dibujan en una ventana, que puede ser la ventana principal de un applet, una ventana hija de un applet, o una ventana de una aplicación independiente. El origen de cada ventana está en la esquina superior izquierda y se identifica como 0,0. Las coordenadas se especifican en píxeles. Todas las salidas sobre una ventana tienen lugar a través de un contexto gráfico. Un contexto gráfico está encapsulado en la clase **Graphics** y se obtiene de dos maneras:

- Se pasa a un applet cuando se llama a alguno de sus métodos, como **paint( )** o **update( ).**
- Es devuelto por el método **getGraphics( )** de **Component.**

En el resto de los ejemplos de este capítulo, los gráficos aparecerán en la ventana principal de un applet. Sin embargo, se pueden aplicar las mismas técnicas para cualquier otra ventana.

En la clase **Graphics** se definen un conjunto de métodos para dibujar. Toda figura se puede dibujar rellena o sólo su borde. Los objetos se dibujan y se rellenan con el color que esté en ese momento seleccionado, por omisión es el negro. Cuando se intenta dibujar un objeto gráfico de dimensiones mayores que las de la ventana, la salida se corta automáticamente. Vamos a ver algunos de los métodos de dibujo.

Dibujar líneas

Las líneas se dibujan con el método **drawLine( ),** que tiene el siguiente formato:

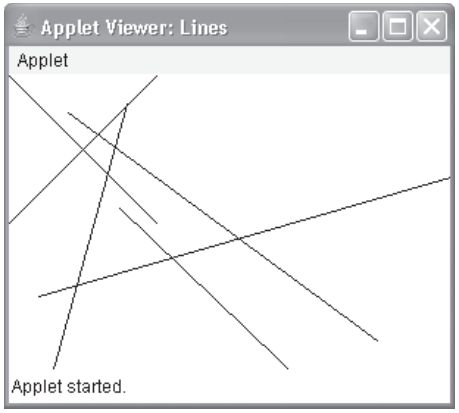
void drawLine(int *x1*, int *y1*, int *x2*, int *y2*)

drawLine( ) dibuja una línea con el color en uso, que comienza en x1, y1 y termina en x2, y2.

El siguiente applet dibuja varias líneas:

```
// Dibujar líneas
import java.awt.*;
import java.applet.*;
/*
<applet code="Lineas" width=300 height=200>
</applet>
*/
public class Lineas extends Applet {
  public void paint(Graphics g) {
    g.drawLine(0, 0, 100, 100);
    g.drawLine(0, 100, 100, 0);
    g.drawLine(40, 25, 250, 180);
    g.drawLine(79, 90, 400, 400);
    g.drawLine(20, l50, 400, 40);
    g.drawLine(5, 290, 80, 19);
  }
}
```

La salida del programa es la siguiente:



Dibujar rectángulos

Los métodos **drawRect( )** y **fillRect( )** dibujan el contorno de un rectángulo y rectángulos rellenos respectivamente. Tienen el siguiente formato:

void drawRect(int *top*, int *left*, int *width*, int *height*)
void fillRect(int *top*, int *left*, int *width*, int *height*)

La esquina superior izquierda del rectángulo está en la posición *top, left*. Las dimensiones del rectángulo las especifican *width* y *height*.

Para dibujar un rectángulo redondeado, se utilizan los métodos **drawRoundRect( )** o **fillRoundRect( )**, que tienen el siguiente formato:

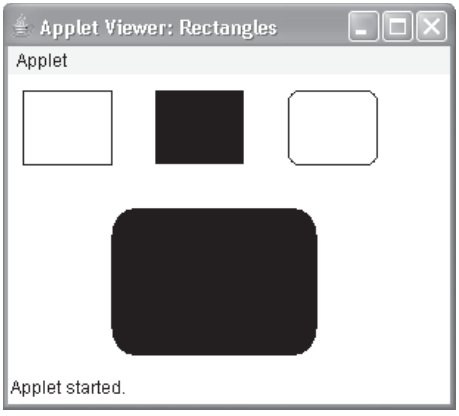
```
void drawRoundRect(int top, int left, int width, int height,
                   int xDiam, int yDiam)
void fillRoundRect(int top, int left, int width, int height,
                   int xDiam, int yDiam)
```

Un rectángulo redondeado es aquel que tiene sus esquinas redondeadas. La esquina superior izquierda del rectángulo está en la posición *top*, *left*. Las dimensiones del rectángulo las especifican *width* y *height*. El diámetro del arco a lo largo del eje X viene dado por *xDiam*, y el diámetro del arco a lo largo del ejeY viene dado por *yDiam*.

El siguiente applet dibuja varios rectángulos:

```
// Dibujar rectángulos
import java.awt.*;
import java.applet.*;
/*
<applet code="Rectangulos" width=300 height=200>
</applet>
*/

public class Rectangulos extends Applet {
  public void paint(Graphics g) {
    g.drawRect(l0, 10, 60, 50);
    g.fillRect(100, 10, 60, 50);
    g.drawRoundRect(190, 10, 60, 50, 15, 15);
    g.fi1lRoundRect(70, 90, 140, 100, 30, 40);
  }
}
```

La salida del programa es la siguiente:



Dibujar elipses y círculos

Para dibujar una elipse se utiliza el método **drawOval( )**, y para una elipse rellena, el método **fillOval( )**. Estos métodos tienen el siguiente formato:

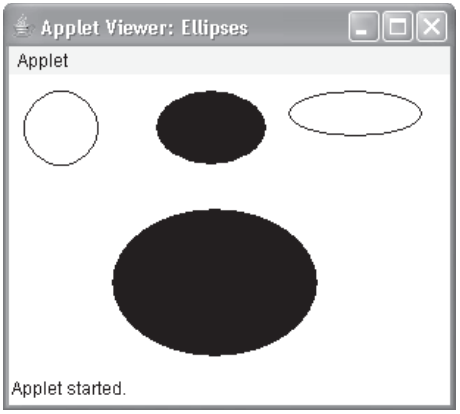
void drawOval(int *top*, int *left*, int *width*, int *height*)
void fillOval(int *top*, int *left*, int *width*, int *height*)

La elipse se dibuja dentro de un rectángulo cuya esquina superior izquierda viene dada por *top*, *left* y cuya anchura y altura están especificadas por *width* y *height*. Para dibujar un círculo, no hay más que especificar las dimensiones de un cuadrado, en lugar de las de un rectángulo.

El siguiente programa dibuja varias elipses:

```
// Dibujar elipses
import java.awt.*;
import java.applet.*;
/*
<applet code="Elipses" width=300 height=200>
</applet>
*/
public class Elipses extends Applet {
  public void paint(Graphics g) {
    g.drawOval(10, 10, 50, 50);
    g.tillOval(100, 10, 75, 50);
    g.drawOval(190, 10, 90, 30);
    g.fillOval(70, 90, 140, 100);
  }
}
```

La salida del programa es la siguiente:



Dibujar arcos

Los arcos se pueden dibujar con los métodos **drawArc( )** y **fillArc( )**:

void drawArc(int *top*, int *left*, int *width*, int *height*, int *startAngle*, int *sweepAngle*) .

void fillArc(int *top*, int *left*, int *width*, int *height*, int *startAngle*, int *sweepAngle*)

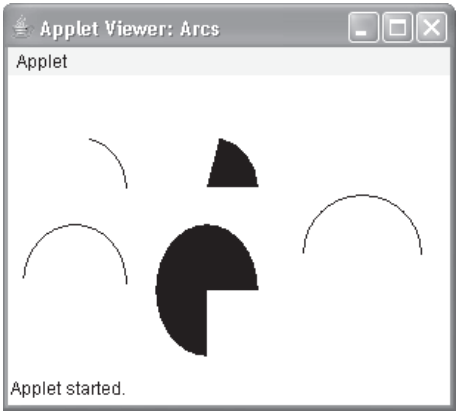
El arco está delimitado en un rectángulo cuya esquina superior izquierda viene dada por top, *left* y cuya anchura y altura son *width* y *height*. El arco se dibuja desde *startAngle* y hasta la

distancia angular dada por *sweepAngle*. Los ángulos se especifican en grados. Los cero grados corresponden con la horizontal, es decir, con las tres en punto de un reloj. El arco se dibuja en el sentido contrario a las agujas del reloj si *sweepAngle* es positivo, y en el sentido de las agujas del reloj si *sweepAngle* es negativo. De esta forma, para dibujar un arco desde las doce en punto a las seis en punto, el ángulo de comienzo sería 90, y la distancia angular 180.

El siguiente applet dibuja varios arcos:

```
// Dibujar arcos
import java.awt.*;
import java.applet.*;
/*
<applet code="Arcos" width=300 height=200>
</applet>
*/

public class Arcos extends Applet {
  public void paint(Graphics g) {
    g.drawArc(10, 40, 70, 70, 0, 75);
    g.fillArc(100, 40, 70, 70, 0, 75);
    g.drawArc(10, 100, 70, 80, 0, 175);
    g.fillArc(100, 100, 70, 90, 0, 270);
    g.drawArc(200, 80, 80, 80, 0, 180);
  }
}
```

La salida de este programa es la siguiente:



Dibujar polígonos

También es posible dibujar figuras con formas arbitrarias utilizando los métodos **drawPolygon( )** y **fillPolygon( )**, que tienen el siguiente formato:

void drawPolygon(int *x*[ ], int *y*[ ], int *numPoints*)
void fillPolygon(int *x*[ ], int *y*[ ], int *numPoints*)

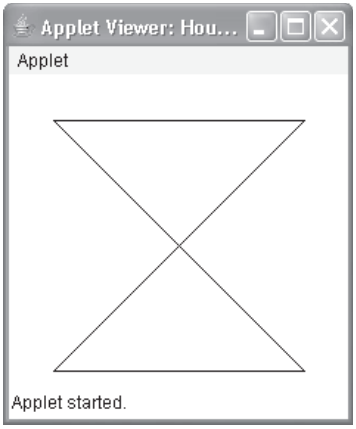
Los vértices del polígono están especificados por las parejas de coordenadas que se encuentran en los arreglos *x, y*. El número de puntos definidos por *x*, y está especificado en *numPoints.*

Estos métodos tienen formas alternativas en las que el polígono está especificado por un objeto **Polygon**.

El siguiente applet dibuja un polígono con forma de reloj de arena:

```
// Dibujar un polígono
import java.awt.*;
import java.applet.*;
/*
<applet code="Reloj" width=230 height=210>
</applet>
*/

public class Reloj extends Applet {
  public void paint(Graphics g) {
    int xPuntos[] = {30, 200, 30, 200, 30};
    int yPuntos[] = {30, 30, 200, 200, 30};
    int num=5;

    g.drawPolygon(xPuntos, yPuntos, num);
  }
}
```

La salida de este programa es la siguiente:



Tamaño de los gráficos

A menudo se quiere ajustar el tamaño del gráfico al de la ventana donde se está dibujando. Para hacer eso, primero hay que obtener las dimensiones actuales de la ventana llamando al método **getSize( )** sobre el objeto ventana. Éste devuelve un objeto de la clase **Dimension** con las dimensiones de la ventana. Una vez que se tienen las dimensiones de la ventana, se puede ajustar el tamaño del gráfico.

Para comprender mejor esta técnica, a continuación se muestra un applet que inicialmente es un cuadrado de 200 × 200 píxeles y va creciendo de 25 en 25 píxeles con cada clic del ratón, tanto en anchura como en altura, hasta llegar a 500 × 500. Cuando mida 500 × 500, el siguiente clic hará que vuelva a medir 200 × 200, y el proceso comenzará otra vez.

Dentro de la ventana, se dibuja un rectángulo que sigue el borde interior de la ventana; y dentro de ese rectángulo, se dibuja una X que ocupa toda la ventana. Este applet trabaja con **appletviewer**, pero no funciona en un navegador.

```
// Ajustar la salida al tamaño actual de la ventana.
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
/*
<applet code="MiApplet" width=200 height=200>
</applet>
*/
public class MiApplet extends Applet {
  final int inc = 25;
  int max = 500;
  int min = 200;
  Dimension d;

  public MiApplet() {
    addMouseListener(new MouseAdapter() {
      public void mouseReleased(MouseEvent yo) {
        int w = (d.width + inc) > max?min :(d.width + inc);
        int h = (d.height + inc) > max?min :(d.height + inc);
        setSize(new Dimension(w, h));
      }
    });
  }
  public void paint(Graphics g) {
    d = getSize ();

    g.drawLine(0, 0, d.width-1, d.height-1);
    g.drawLine(0, d.height-1, d.width-1, 0);
    g.drawRect(0, 0, d.width-1, d.height-1);
  }
}
```

Trabajar con color

Java trabaja con los colores de manera portátil e independiente del dispositivo. El sistema de color de AWT permite especificar cualquier color. Para ello se busca el color que mejor se ajuste al solicitado, teniendo en cuenta las limitaciones del hardware de visualización en el que se está ejecutando el programa o el applet. De esta manera, el código no tiene que preocuparse sobre cómo trata cada dispositivo gráfico a los colores. El color se encapsula en la clase **Color**.

Como se vio en el Capítulo 21, la clase **Color** define varias constantes (por ejemplo, **Color.black**) para especificar un conjunto de colores comunes. También es posible crear colores propios utilizando uno de los constructores de la clase Color. Las formas que más se utilizan son las siguientes:

Color(int *red*, int *green*, int *blue*)
Color(int *rgbValue)*
Color(float *red*, float *green*, *float blue*)

El primer constructor toma tres enteros que especifican el color como una mezcla de rojo, verde y azul. Esos valores deben estar entre 0 y 255, como en el siguiente ejemplo:

```
new Color(255, 100, 100); // rojo claro.
```

El segundo constructor de color toma un solo entero que ya contiene la mezcla del rojo, verde y azul. En este entero, el rojo está entre los bits 16 y 23, el verde entre el 8 y el 15, y el azul entre los bits 0 y 7. A continuación se muestra un ejemplo de este constructor:

```
int nuevoRojo = (0xff000000 | (0xc0 << 16) | (0x00 << 8) | 0x00);
Color rojoOscuro = new Color(nuevoRojo);
```

El último constructor, **Color (float, float, float)**, toma tres valores flotantes (entre 0.0 y 1.0) que especifican la mezcla del rojo, verde y azul.

Una vez que se ha creado el color, lo podemos utilizar para establecer el color del frente y/o el color fondo utilizando los métodos **setForeground( )** y **setBackground( )** descritos en el Capítulo 21. También es posible seleccionar el color creado como el color de dibujo actual.

Métodos de la clase Color

La clase Color define varios métodos para manipular colores. A continuación se analizan cada uno de ellos.

Uso del tono (Hue), la saturación (Saturation) y el brillo (Brightness)

El modelo de color tono-saturación-brillo (HSB por sus siglas en inglés) es una alternativa al modelo rojo-verde-azul (RGB por sus siglas en inglés) que acabamos de ver para especificar colores. De manera figurada, el tono es una rueda de colores. El tono se establece con un número comprendido entre 0.0 y 1.0 (los colores aproximadamente son: rojo, naranja, amarillo, verde, azul, índigo y violeta). La saturación es otra escala comprendida entre 0.0 y 1.0, y va desde tonos pasteles claros hasta tonos intensos. El brillo también toma valores desde 0.0 a 1.0, donde 1 es el blanco brillante y 0 es el negro. La clase Color proporciona dos métodos para cambiar entre RGB y HSB:

static int HSBtoRGB(float *hue*, float *saturation*, float *brightness*)
static float[ ] RGBtoHSB(int *red*, int *green*, int blue, float *values*[ ])

HSBtoRGB( ) devuelve un valor RGB compatible con el constructor **Color(int)**. **RGBtoHSB( )** devuelve un arreglo de tipo float con valores HSB que corresponden a los enteros RGB. Si values no es **null**, entonces el arreglo tiene los valores HSB y es devuelto. En caso contrario, se crea un nuevo arreglo y los valores HSB se devuelven en él. En ambos casos, el arreglo contiene el tono en la posición 0, la saturación en la posición 1 y el brillo en la posición 2.

getRed( ), getGreen( ) y getBlue( )

Se pueden obtener las componentes rojo, verde y azul de un color utilizando los métodos **getRed( )**, **getGreen( )** y **getBlue( )**, cuya forma general es:

int getRed( )
int getGreen( )
int getBlue( )

Cada uno de estos métodos devuelve, en los 8 bits más bajos de un entero, la componente RGB utilizada al invocar al objeto **Color**.

getRGB( )

Para obtener la representación RGB de un color, se puede utilizar el método **getRGB( )** cuyo formato es:

int getRGB( )

El valor devuelto está organizado como se describió anteriormente.

Establecer el color para los gráficos

Por omisión, los objetos gráficos se dibujan con el color actual del frente. Se puede cambiar este color llamando al método **setColor( )** de **Graphics**:

void setColor(Color *nuevoColor*)

Aquí, nuevoColor especifica el nuevo color de dibujo.

Se puede obtener el color en uso llamando al método **getColor( ),** que tiene el siguiente formato:

Color getColor( )

Un ejemplo de applet con colores

El siguiente applet construye varios colores y dibuja varios objetos utilizando esos colores:

```
// Ejemplo con colores.
import java.awt.* ;
import java.applet.*;
/*
<applet code = "ColorDemo" width=300 height=200>
</applet>
*/

public class ColorDemo extends Applet {
  // dibujo de líneas
  public void paint(Graphics g) {
    Color c1 = new Color(255, 100, 100);
    Color c2 = new Color(l00, 255, 100);
    Color c3 = new Color(l00, 100, 255);

    g.setColor (c1) ;
    g.drawLine(0, 0, 100, 100);
    g.drawLine(0, 100, 100, 0);

    g.setColor(c2);
    g.drawLine(40, 25, 250, 180);
    g.drawLine(75, 90, 400, 400);

    g.setColor(c3);
    g.drawLine(20, 150, 400, 40);
    g.drawLine(5, 290, 80, 19};

    g.setColor(Color.red);
    g.drawOval(10, 10, 50, 50);
    g.fillOval(70, 90, 140, 100);
```

```
    g.setColor(Color.blue);
    g.drawOval(190, 10, 90, 30);
    g.drawRect(10, 10, 60, 50);

    g.setColor(Color.cyan);
    g.fillRect(100, 10, 60, 50);
    g.drawRoundRect(190, 10, 60, 50, 15, 15);
  }
}
```


Establecer el modo de pintado

El *modo de pintar* determina cómo se dibujan los objetos en una ventana. Por omisión, cada nueva salida que se realiza a una ventana se sobrepone al contenido de lo que hubiese en la anterior. Sin embargo, es posible tener sobre la ventana nuevos objetos en modo XOR utilizando el método **setXORMode( )**, que tiene el formato siguiente:

void setXORMode(Color *xorColor*)

donde *xorColor* especifica el color que se utilizará para hacer el XOR en la ventana cuando se dibuja un objeto. La ventaja del modo XOR es que se garantiza que el nuevo objeto siempre esté visible, cualquiera que sea el color sobre el que se dibuje el objeto.

Para volver al modo de sobreescritura, hay que llamar al método **setPaintMode( ):**

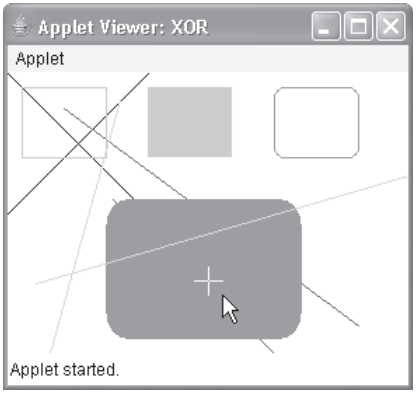
void setPaintMode( )

En general, se utilizará el modo de sobreescritura para las salidas normales, y el modo XOR para situaciones especiales. Por ejemplo, el siguiente programa muestra una cruz que se mueve con el ratón. La cruz está en modo XOR sobre la ventana y siempre está visible, independientemente del color sobre el que se dibuje.

```
// Ejemplo del modo XOR.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="XOR" width=400 height=200>
  </applet>
*/

public class XOR extends Applet {
  int chsX=l00, chsY=l00;

public XOR() {
  addMouseMotionListener(new MouseMotionAdapter() {
    public void mouseMoved(MouseEvent yo) {
      int x = yo.getX();
      int y = yo.getY();
      chsX = x-l0;
      chsY = y-l0;
      repaint();
    }
  });
}
```

```
  public void paint(Graphics g) {
    g.drawLine(0, 0, 100, 100);
    g.drawLine(0, 100, 100, 0);
    g.setColor(Color.blue);
    g.drawLine(40, 25, 250, 180);
    g.drawLine(75, 90, 400, 400);
    g.setColor(Color.green);
    g.drawRect(l0, 10, 60, 50);
    g.fillRect(l00, 10, 60, 50);
    g.setColor(Color.red);
    g.drawRoundRect(190, 10, 60, 50, 15, 15);
    g.fillRoundRect(70, 90, 140, 100, 30, 40);
    g.setColor(Color.cyan) ;
    g.drawLine(20, 150, 400, 40);
    g.drawLine(5, 290, 80, 19);

    // apuntador en modo xor
    g.setXORMode(Color.black);
    g.drawLine,(chsX-10 chsY, chsX+10, chsY);
    g.drawLine(chsX, chsY-10, chsX, chsY+10);
    g.setPaintMode();
  }
}
```

A continuación se muestra la salida de este programa:



Trabajando con tipos de letra

AWT permite trabajar con múltiples tipos de letra, que han surgido de los tipos tradicionales de imprenta y se han convertido en una parte importante de la visualización de documentos generados por computadora. AWT proporciona gran flexibilidad para la manipulación y selección dinámica de tipos de letra.

Los tipos de letra tienen un nombre de familia, un nombre lógico, y un nombre de apariencia. El *nombre de familia* es el nombre general del tipo de letra, como, por ejemplo, Courier. El *nombre lógico* especifica una categoría del tipo de letra, como Monospaced. El *nombre de apariencia* especifica un tipo de letra en particular, como Courier Italic.

La clase **Font** encapsula los tipos de letra. En la Tabla 23-2 se muestran varios de los métodos definidos por **Font**.

Método	Descripción
static Font decode(String str)	Devuelve un tipo de letra dando su nombre.
boolean equals (Object *FontObj*)	Devuelve **true** si el objeto que llama al método contiene el mismo tipo de letra que el especificado por *FontObj*. En caso contrario, devuelve **false**.
String getFamily( )	Devuelve el nombre de la familia a la que pertenece el tipo de letra que ha realizado la llamada.
static Font getFont (String *property*)	Devuelve el tipo de letra asociado a la propiedad especificada por *property*. Devuelve **null** si no existe *property*.
static Font getFont (String *property*, Font *defaultFont*)	Devuelve el tipo de letra asociado a la propiedad especificada por *property*. Devuelve el tipo de letra especificado por *defaultFont* si no existe *property*.
String getFontName()	Devuelve el nombre de apariencia del tipo de letra que ha llamado al método.
String getName( )	Devuelve el nombre lógico del tipo de letra que ha llamado al método.
int getSize( )	Devuelve el tamaño, en puntos, del tipo de letra que llama al método.
int getStyle( )	Devuelve el valor de estilo del tipo de letra que llama al método.
int hashCode( )	Devuelve el código asociado al objeto que llama al método.
boolean isBold( )	Devuelve **true** si el tipo de letra incluye el estilo **BOLD**. En caso contrario, devuelve **false**.
boolean isItalic( )	Devuelve **true** si el tipo de letra incluye el estilo **ITALIC**. En caso contrario, devuelve **false**.
boolean isPlain( )	Devuelve **true** si el tipo de letra incluye el estilo **PLAIN**. En caso contrario, devuelve **false**.
String toString( )	Devuelve la cadena equivalente al tipo de letra que llama al método.

TABLA 23-2 Algunos métodos definidos por la clase **Font**

La clase **Font**, define las siguientes variables:

Variable	Significado
String name	Nombre del tipo de letra
float pointSize	Tamaño, en puntos, del tipo de letra
int size	Tamaño, en puntos, del tipo de letra
int style	Estilo del tipo de letra

Determinación de los tipos de letra disponibles

Cuando se trabaja con tipos de letra suele ser necesario saber qué tipos de letra están disponibles en la computadora. Para obtener esa información, se utiliza el método **getAvailableFontFamilyNames( )** definido por la clase **GraphicsEnvironment**. La forma general del método es:

String[ ] getAvailableFontFamilyNames( )

Este método devuelve un arreglo de cadenas con los nombres de las familias de tipos de letra disponibles.

La clase GraphicsEnvironment también define al método **getAllFonts( )**:

```
Font[ ] getAllFonts( )
```

Este método devuelve un arreglo de objetos **Font** que contiene todos los tipos de letra disponibles.

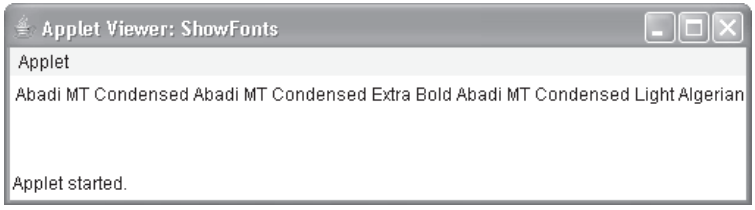
Como estos métodos son miembros de la clase **GraphicsEnvironment** es necesario contar con una referencia **GraphicsEnvironment** para llamarlos. Se puede obtener esta referencia utilizando el método estático **getLocalGraphicsEnvironment( ),** que está definido por **GraphicsEnvironment**:

```
static GraphicsEnvironment getLocalGraphicsEnvironment( )
```

A continuación se muestra un applet que obtiene los nombres de las familias de tipos de letra disponibles:

```
// Ver los tipos de letra disponibles
/*
<applet code="ShowFonts" width=550 height=60>
</applet>
*/
import java.applet.*;
import java.awt.*;

public class ShowFonts extends Applet {
  public void paint(Graphics g) {
    String msg = "";
    String FontList[];

    GraphicsEnvironment ge =
      GraphicsEnvironment.getLocalGraphicsEnvironment();
    FontList = ge.getAvailableFontFamilyNames();
    for(int i = 0; i < FontList.length; i++)
      msg += FontList[i] + " ";

    g.drawString(msg, 4, 16);
  }
}
```

A continuación se muestra la salida de este programa. No obstante, cuando usted lo ejecute, seguramente verá una lista de tipos de letra diferente de la que aquí aparece.



Creación y selección de un tipo de letra

Para seleccionar un nuevo tipo de letra, primero hay que construir un objeto **Font** que describa ese tipo de letra. Una de las formas del constructor **Font** es:

Font (String *fontName*, int *fontStyle*, int *pointSize*)

Aquí, *fontName* especifica el nombre del tipo de letra deseado. Se puede especificar el nombre utilizando tanto el nombre lógico como el nombre de apariencia. Todos los entornos Java admiten los siguientes tipos de letra: Dialog, DialogInput, Sans Serif, Serif y Monospaced. Dialog es el tipo de letra que utilizan las cajas de diálogo del sistema. También es el tipo de letra que se usa por omisión. Se puede utilizar cualquier otro tipo de letra que exista en la computadora, pero hay que tener cuidado con eso puesto que puede que ese otro tipo de letra no esté disponibles en todos los entornos.

El estilo del tipo de letra se especifica en fontStyle. El estilo está formado por una o más de estas tres constantes: **Font.PLAIN, Font.BOLD** y **Font.ITALIC**. Los estilos se combinan utilizando la operación OR. Por ejemplo, **Font.BOLD | Font.ITALIC** especifica un estilo en negrita y cursiva.

El tamaño del tipo de letra, en puntos, se especifica con *pointSize*.

Para utilizar un tipo de letra propio, se utiliza el método **setFont( )**, que está definido en la clase **Component**, y tiene la siguiente forma general:

void setFont(*Font fontObj)*

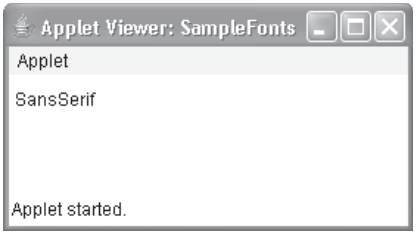
donde *fontObj* es el objeto que contiene el tipo de letra deseado.

El siguiente programa muestra un ejemplo de cada tipo de letra estándar. Cada vez que se haga clic con el ratón en la ventana, se seleccionará un tipo de letra y se visualizará su nombre.

```
// Mostrar los tipos de letra.
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
/*
  <applet code="EjemploFonts" width=200 height=100>
  </applet>
*/

public class EjemploFonts extends Applet {
  int next = 0;
  Font f;
  String msg;
  public void init() {
    f = new Font("Dialog", Font. PLAIN, 12);
    msg = "Dialog";
    setFont(f);
    addMouseListener(new MiMouseAdapter(this));
  }

  public void paint(Graphics g) {
    g.drawString(msg, 4, 20);
  }
}
```

```
class MiMouseAdapter extends MouseAdapter {
  EjemploFonts ejemploFonts;
  public MiMouseAdapter(EjemploFonts ejemploFonts) {
    this.ejemploFonts = ejemploFonts;
  }
  public void mousePressed(MouseEvent yo) {
    // cambia el tipo de letra en cada clic del ratón
    ejemploFonts.next++;
    switch(ejemploFonts.next) {
    case 0:
      ejemploFonts.f = new Font("Dialog", Font.PLAIN, 12);
      ejemploFonts.msg = "Dialog";
      break;
    case 1:
      ejemploFonts.f = new Font("DialogInput", Font.PLAIN, 12);
      ejemploFonts.msg = "DialogInput";
      break;
    case 2:
      ejemploFonts.f = new Font("SansSerif", Font.PLAIN, 12);
      ejemploFonts.msg ="SansSerif";
      break;
    case 3:
      ejemploFonts.f = new Font("Serif", Font.PLAIN, 12);
      ejemploFonts.msg = "Serif";
      break;
    case 4:
      ejemploFonts.f = new Font("Monospaced", Font.PLAIN, 12);
      ejemploFonts.msg = "Monospaced";
      break;
    }
    if(ejemploFonts.next == 4) ejemploFonts.next =-1;
    ejemploFonts.setFont(ejemploFonts.f);
    ejemploFonts.repaint();
  }
}
```

Un ejemplo de la salida del programa es el siguiente:



Información sobre los tipos de letra

Supongamos que se quiere obtener información sobre el tipo de letra seleccionado. Primero hay que obtener el tipo de letra en curso llamando al método **getFont( )**. Este método está definido por la clase **Graphics**, como se muestra a continuación:

Font getFont( )

Una vez que se tiene el tipo de letra que se está utilizando, se puede obtener información de él utilizando varios métodos definidos por la clase **Font**. Por ejemplo, este applet visualiza el nombre, familia, tamaño y estilo del tipo de letra seleccionado:

```
// Visualizar información del tipo de letra.
import java.applet.*;
import java.awt.*;
/*
<applet code="FontInfo" width=350 height=60>
</applet>
*/

public class FontInfo extends Applet {
  public void paint(Graphics g) {
    Font f = g.getFont();
    String fontName = f.getName();
    String fontFamily = f.getFamily();
    int fontSize = f.getSize();
    int fontStyle = f.getStyle();

    String msg = "Familia: " + fontName;
    msg += ", Tipo de letra: " + fontFamily;
    msg += ", Tamaño: " + fontSize + ", Estilo: ";
    if((fontStyle & Font.BOLD) == Font.BOLD)
      msg += "Bold ";
    if((fontStyle & Font.ITALIC) == Font.ITALIC)
      msg += "Italic ";
    if((fontStyle & Font.PLAIN) == Font.PLAIN)
      msg += "Plain ";

    g.drawString (msg, 4, 16);
  }
}
```

Gestión de la salida de texto utilizando FontMetrics

Como se ha explicado, Java admite varios tipos de letra. En la mayoría de ellos, todos los caracteres no tienen el mismo tamaño, ya que la mayoría de los tipos de letra son proporcionales. Además, la altura de cada carácter, la longitud de los *descensos* (las partes que cuelgan de letras, como en la *y*) y la cantidad de espacio entre líneas horizontales varían de un tipo de letra a otro. Más aún, puede cambiar hasta el tamaño del punto de la letra. En realidad, el hecho de que estos (y otros) atributos sean variables no tiene más consecuencia que Java solicite al programador que gestione manualmente el texto de salida.

Como el tamaño de cada tipo de letra puede ser diferente y como se pueden cambiar los tipos de letra mientras se ejecuta el programa, debe haber alguna forma de determinar las dimensiones y otra serie de atributos del tipo de letra seleccionado en un momento dado. Por ejemplo, escribir una línea de texto después de otra implica que es necesario conocer la altura del tipo de letra y cuantos píxeles tiene que haber entre las líneas. Para esto, AWT incorpora la clase **FontMetrics**, que encapsula información relacionada con el tipo de letra. Comencemos definiendo la terminología común que se utiliza al describir los tipos de letra:

Altura	Es el tamaño, desde el punto superior al inferior, del carácter más alto en ese tipo de letra.
Línea base	Es la línea sobre la que están alineadas las partes inferiores de los caracteres (sin tener en cuenta los descensos).
Ascenso	Es la distancia desde la línea base al punto superior de un carácter.
Descenso	La distancia desde la línea base al punto más bajo de un carácter.
Interlineado	Es la distancia entre la parte inferior de una línea de texto y la parte superior de la siguiente línea.

Como sabemos, se ha utilizado el método **drawString( )** en muchos de los ejemplos anteriores. Este método pinta una cadena con el tipo de letra y color en curso, comenzando en una posición dada. Sin embargo, esta posición está en la parte izquierda de la línea base de los caracteres, y no en la esquina superior izquierda como suele ser normal en otros métodos de dibujo. Es un error habitual dibujar una cadena en la misma coordenada en la que se dibujaría un cuadro. Por ejemplo, si se ha dibujado un rectángulo en la coordenada 0,0 del applet, se verá un rectángulo completo; pero si se ha dibujado el texto"proyecto gigante"en 0,0, sólo se verán los extremos inferiores (o descensos) de la *g* y la *p*. Como se verá más adelante, utilizando FontMetrics se puede determinar la posición adecuada para cada cadena que se desee visualizar.

FontMetrics define varios métodos que ayudan a gestionar la salida de texto. En la Tabla 23-3 se muestran los más utilizados. Estos métodos ayudan a visualizar texto de forma adecuada en una ventana. Veamos algunos ejemplos.

Método	Descripción
int bytesWidth(byte b[ ], int *start*, int *numBytes*)	Devuelve el ancho de los *numBytes* caracteres contenidos en el arreglo *b* comenzando en *start*.
int charWidth(char c[ ], int *start*, int *numChars*)	Devuelve el ancho de los *numChars* caracteres contenidos en el arreglo *c* comenzando en *start*.
int charWidth(char *c*)	Devuelve el ancho de *c*.
int charWidth(int *c*)	Devuelve el ancho de *c*.
int getAscent( )	Devuelve el ascenso del tipo de letra.
int getDescent( )	Devuelve el descenso del tipo de letra.
Font getFont( )	Devuelve el tipo de letra.
int getHeight( )	Devuelve la altura de una línea de texto. Este valor se puede utilizar para escribir varias líneas de texto en una ventana.
int getLeading( )	Devuelve el espacio que hay entre dos líneas de texto.
int getMaxAdvance( )	Devuelve el ancho del carácter más ancho. Devuelve –1 si este valor no está disponible.
int getMaxAscent( )	Devuelve el máximo ascenso.
int getMaxDescent( )	Devuelve el máximo descenso.
int[ ] getWidths( )	Devuelve los anchos de los primeros 256 caracteres.
int stringWidth(String *str*)	Devuelve el ancho de la cadena especificada por *str*.
String toString( )	Devuelve la cadena equivalente del objeto que llama al método.

TABLA 23-3 Algunos métodos definidos por la clase **FontMetrics**

Visualización de varias líneas de texto

Tal vez el uso más común de **FontMetrics** sea determinar el espacio entre líneas de texto. También se suele utilizar para determinar la longitud de una cadena que se está visualizando. Vamos a ver cómo se realizan estas tareas.

En general, para visualizar varias líneas de texto, el programa tiene que seguir la pista, manualmente, de la posición donde se realiza la salida. Cada vez que se desee una nueva línea, la coordenada Y tiene que avanzar al principio de la nueva línea; y cada vez que se visualice una cadena la coordenada X tiene que ir al punto donde termina la cadena. Esto permite escribir la siguiente cadena justo después de la anterior.

Para determinar el espacio entre líneas, se puede utilizar el valor devuelto por **getLeading( )**. Para determinar la altura total del tipo de letra, hay que sumar al valor devuelto por **getAscent( )** el valor devuelto por **getDescent( )**. Estos valores se pueden utilizar para establecer la posición de cada línea de texto. Sin embargo, muchas veces no hará falta utilizar esos valores individualmente, ya que normalmente bastará con conocer la altura total de una línea, que es la suma del espacio adicional y de los valores de ascenso y descenso del tipo de letra. La forma más sencilla de obtener este valor es llamar a **getHeight( )**. Sólo hay que incrementar la coordenada Y en ese valor cada vez que se quiera avanzar a una nueva línea de texto.

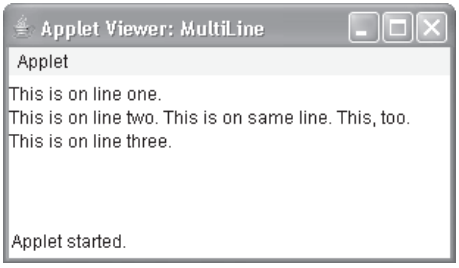
Cuando se ha sacado algo de texto y se quiere continuar en esa misma línea, se debe conocer la longitud, en píxeles, de cada cadena que se quiera visualizar. Para obtener este valor, hay que llamar a **stringWidth( )**. Este valor se utiliza para añadirlo a la coordenada X cada vez que se visualiza una línea.

En el siguiente applet se muestra cómo sacar varias líneas de texto en una ventana y como visualizar varias sentencias en una misma línea. Nótese que las variables **curX** y **curY** mantienen la posición donde se está escribiendo el texto.

```
// Escribir texto en varias líneas.
import java.applet.*;
import java.awt.*;
/*
<applet code="MultiLinea" width=300 height=100>
</applet>
*/

public class MultiLinea extends Applet {
  int curX=0, curY=0; // posición actual

  public void init() {
    Font f = new Font("SansSerif", Font.PLAIN, 12);
    setFont(f);
  }
  public void paint(Graphics g) {
    FontMetrics fm = g.getFontMetrics();

    nextLine("Ésta es la línea uno.", g);
    nextLine("Ésta es la línea dos.", g);
    sameLine(" Ésta es la misma línea.", g);
    sameLine(" Ésta, también.", g);
    nextLine("Ésta es la línea tres.", g);
  }
```

```
  // Avanzar a otra línea.
  void nextLine(String s, Graphics g) {
    FontMetrics fm = g.getFontMetrics();

    curY += fm.getHeight(); // avanzar a la siguiente línea
    curX = 0;
    g.drawString(s, curX, curY);
    curX = fm.stringWidth(s); // avanzar al final de la línea
  }

  // Mostrar la misma línea.
  void sameLine(String s, Graphics g) {
    FontMetrics fm = g.getFontMetrics();

    g.drawString(s, curX, curY);
    curX += fm.stringWidth(s); // avanzar al final de la línea
  }
}
```

A continuación se muestra un ejemplo de la ejecución de este programa:



Centrar el texto

A continuación se muestra un ejemplo que centra el texto en una ventana, tanto horizontal como verticalmente. Obtiene el ascenso, descenso y anchura del texto y calcula la posición donde se debe visualizar para que esté centrado.

```
// Centrar texto.
import java.applet.*;
import java.awt.*;
/*
  <applet code="TextoCentrado" width=200 height=l00>
  </applet>
*/

public class TextoCentrado extends Applet {
  final Font f = new Font("SansSerif", Font.BOLD, 18);

  public void paint(Graphics g) {
    Dimension d = this.getSize();
    g.setColor(Color.white);
    g.fillRect(0, 0, d.width, d.height);
    g.setColor(Color.black);
    g.setFont(f);
    dibujarStringCentrado("Esto está centrado.", d.width, d.height, g);
```

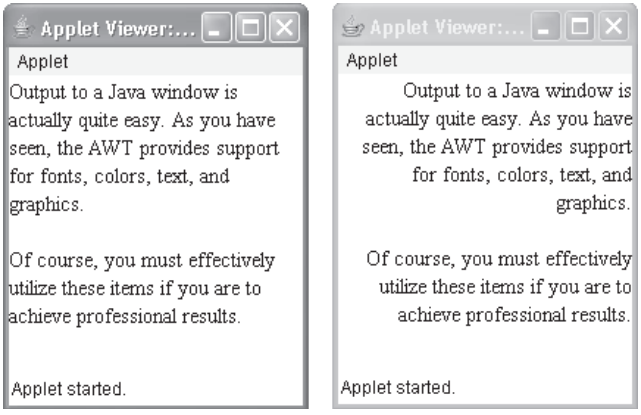
```
    g.drawRect(0, 0, d.width-1, d.height-1);
  }

  public void dibujarStringCentrado(String s, int w, int h, Graphics g)
    FontMetrics fm = g.getFontMetrics();
    int x = (w - fm.stringWidth(s))/2;
    int y = (fm.getAscent() + (h - (fm.getAscent()
    + fm.getDescent()))/2);
    g.drawString(s, x, y);
  }
}
```

Un ejemplo de la ejecución de este programa es el siguiente:

Alineamiento de varias líneas de texto

Si alguna vez se ha utilizado un procesador de textos, se habrá visto que el texto se puede alinear en uno o los dos extremos sobre una hipotética línea recta. Por ejemplo, la mayoría de los procesadores de texto pueden centrar el texto, o justificarlo a la izquierda, a la derecha o a ambos lados. Veremos en el siguiente programa cómo realizar todo esto.

En el programa se divide la cadena que se va a justificar en palabras individuales. El programa guarda, para cada palabra, su longitud con el tipo de letra en curso y pasa automáticamente a la siguiente línea si la palabra no va a caber en la línea en curso. Cada línea que se completa se visualiza en la ventana con el estilo de alineamiento seleccionado. Cada vez que se haga clic con el ratón en la ventana del applet, se cambia el estilo de alineamiento. A continuación se muestra un ejemplo de la salida de este programa:



PARTE II

```
// Mostrar texto alineado.
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
/* <title>Text Layout</title>
    <applet code="TextLayout" width=200 height=200>
    <param name="text" value="Sacar algo a una ventana Java es realmente fácil.
      Como se ha visto, el AWT proporciona los medios necesarios
      para trabajar con tipos de letra, colores, texto, y gráficos. <P> Por su-
      puesto,
      si se quieren obtener resultados profesionales,
      se deben dominar estos elementos.">
    <param name="fontname" value="Serif">
    <param name="fontSize" value="14">
  </applet>
*/

public class TextLayout extends Applet {
  final int LEFT = 0;
  final int RIGHT = 1;
  final int CENTER = 2;
  final int LEFTRIGHT = 3;
  int align;
  Dimension d;
  Font f;
  FontMetrics fm;
  int fontSize;
  int fh, bl;
  int space;
  String text;

  public void init() {
    setBackground(Color.white);
    text = getParameter("text");
    try {
      fontSize = Integer.parseInt(getParameter("fontSize"));
    } catch (NumberFormatException e) {
      fontSize=14;
    }
    align = LEFT;
    addMouseListener(new MiMouseAdapter(this));
  }

  public void paint(Graphics g) {
    update(g) ;
  }

  public void update(Graphics g) {
    d = getSize();
    g.setColor(getBackground());
    g.fillRect(0, 0, d.width, d.height);
    if(f==null) f = new Font(getParameter("fontname"),
                             Font.PLAIN, fontSize);
```

```
    g. setFont (f);
    if(fm == null) {
        fm = g.getFontMetrics();
        bl = fm.getAscent();
        fh = bl + fm.getDescent() ;
        space= fm.stringWidth(" ");
    }

    g.setColor(Color.black);
    StringTokenizer st = new StringTokenizer(text);
    int x = 0;
    int nextx;
    int y = 0;
    String word, sp;
    int wordCount = 0;
    String line = "";
    while (st.hasMoreTokens()) {
      word = st.nextToken();
      if(word.equals("<P>")) {
        drawString(g, line, wordCount,
                   fm.stringWidth(line), y+bl);
      line = "";
      wordCount= 0;
      x = 0;
      y=y+ (fh*2);
    }
    else {
      int w = fm.stringWidth(word);
      if(( nextx = (x+space+w)) > d.width ) {
        drawString(g, line, wordCount,
                   fm.stringWidth(line), y+bl);
        line = "";
        wordCount=0
        x = 0;
        y = y + fh;
      }
      if(x!=0) {sp = " ";} else {sp = "";}
      line = line + sp + word;
      x = x + space + w;
      wordCount++;
    }
  }
  drawString(g, line, wordCount, fm.stringWidth(line), y+bl);
}

public void drawString(Graphics g, String line,
                       int wc, int lineW, int y) {
  switch (align) {
    case LEFT: g.drawString(line, 0 y);
      break;
    case RIGHT: g.drawString(line, d.width-lineW, y);
      break;
    case CENTER: g.drawString(line, (d.width-lineW)/2, y);
      break;
```

```
      case LEFTRIGHT:
        if(lineW < (int) (d.width*.75)) {
          g.drawString(line, 0, y);
        }
        else {
          int toFill = (int) (d.width - lineW)/wc;
          int nudge = d.width - lineW - (toFill*wc);
          int s = fm.stringWidth(" ");
          StringTokenizer st = new StringTokenizer(line);
          int x = 0
          while(st.hasMoreTokens()) {
            String word = st.nextToken();
            g.drawString(word, x, y);
            if(nudge>0) {
              x = x + fm.stringWidth(word) + space + toFill + 1;
              nudge--;
            } else {
              x = x + fm.stringWidth(word) + space + toFill;
            }
          }
        }
        break;
      }
  }
}
class MiMouseAdapterextends MouseAdapter {
  TextLayout t1;
  public MiMouseAdapter(TextLayout t1) {
    this.t1 = t1;
  }
  public void mouseClicked(MouseEvent yo) {
    tl.align = (tl.align + 1) % 4;
    t1.repaint();
  }
}
```

Veamos con más detalle cómo funciona este applet. Primero, el applet crea varias constantes que se utilizarán para determinar el estilo de alineamiento, y después declara diversas variables. El método **init( )** obtiene el texto que se va a mostrar e inicializa el tamaño del tipo de letra en un bloque **try-catch**, que establece el tamaño del tipo de letra a 14 si se ha omitido el parámetro **fontSize** en el HTML. El parámetro text es una larga cadena de texto que utiliza a la etiqueta HTML **<P>** como separador de párrafos.

El método **update( )** es el motor de este ejemplo. Establece el tipo de letra y obtiene la línea base y la altura del tipo de letra a partir de un objeto de la clase FontMetrics. Después, crea un **StringTokenizer** y lo utiliza para obtener la siguiente palabra (separada por espacios en blanco) de la cadena dada en **text**. Si la siguiente palabra es **<P>**, avanza verticalmente. En caso contrario, **update( )** revisa si la longitud de esa palabra con el tipo de letra actual cabe en lo que queda de línea. Si la línea ya está completa o si no hay más palabras, se muestra la línea con el método **drawString( )**.

Los tres primeros casos de **drawString( )** son simples. Cada uno de ellos alinea, la cadena que está en el parámetro, a la izquierda, a la derecha o en el centro de la columna, dependiendo del estilo de alineamiento. El caso **LEFTRIGHT** alinea los dos lados, izquierdo y derecho, del texto. Esto significa que se necesita calcular el espacio en blanco que queda (la diferencia entre la anchura de la cadena y la anchura de la columna) y distribuir ese espacio entre cada una de las palabras. El último método de esta clase cambia el estilo de alineamiento cada vez que se hace clic con el ratón en la ventana del applet.

CAPÍTULO 24

AWT: controles, gestores de organización y menús

En este capítulo, continua nuestra exploración del conjunto de herramientas gráficas, AWT (Abstract Window Toolkit). El capítulo comienza con el estudio de los controles estándar y los gestores de organización definidos por Java. Luego se estudian los menús y las barras de menú. Este capítulo incluye además un análisis de dos componentes de alto nivel: el cuadro de diálogo y el cuadro de diálogo para archivos. El capítulo concluye profundizando en la gestión de eventos.

Los *controles* son componentes que permiten al usuario interactuar con la aplicación de varias maneras. Por ejemplo, el control más utilizado es el botón. Un *gestor de organización*, posiciona automáticamente los componentes dentro de un contenedor. Por tanto, la apariencia de una ventana está determinada por la combinación de controles que contiene, y por el gestor de organización utilizado para colocarlos.

Además de los controles, una ventana también puede incluir una *barra de menú* estándar. Cada entrada en la barra del menú, activa un menú desplegable con opciones que el usuario puede seleccionar. La barra del menú siempre se encuentra en la parte superior de la ventana. Aunque tengan diferente aspecto, las barras de menú se gestionan de la misma forma que el resto de los controles.

Aunque es posible situar manualmente los componentes dentro de una ventana, resulta bastante tedioso. El gestor de organización automatiza esta tarea. En la primera parte del capítulo, donde se introducen varios controles, se utiliza el gestor de organización por omisión. Este gestor muestra los componentes en un contenedor utilizando la organización de izquierda a derecha y de arriba abajo. Una vez que se han analizado los controles, se estudian los gestores de organización, y cuál es la mejor manera de posicionar los controles.

Conceptos básicos de los controles

AWT soporta los siguientes tipos de controles:

- Etiquetas
- Botones
- Checkbox
- Listas de opciones
- Listas

- Barras de desplazamiento
- Cuadros de texto

Estos controles son subclases de **Component**.

Añadir y eliminar controles

Para incluir un control en una ventana es necesario añadirlo a la ventana. Para hacer esto, primero hay que crear una instancia del control deseado y después añadirlo a la ventana llamando al método **add( ),** que está definido en la clase **Container**. El método **add( )** tiene varios formatos. El siguiente formato es el que se utiliza en la primera parte de este capítulo:

Component add(Component *compObj*)

Donde *compObj* es una instancia del control que se quiere añadir. El método devuelve una referencia a *compObj*. Una vez que se ha añadido el control, será visible automáticamente siempre que se muestre su ventana padre.

Algunas veces, cuando ya no se necesita utilizar un control, se puede querer eliminar dicho control de la ventana. Para hacerlo, hay que llamar al método **remove( ).** Este método también está definido en la clase **Container**. Su forma general es la siguiente:

void remove(Component *obj*)

Donde *obj* es una referencia al control que se quiere eliminar. Se pueden eliminar todos los controles llamando al método **removeAll( )**.

Responder a los controles

Con excepción de las etiquetas, que son controles pasivos, todos los demás controles generan eventos cuando el usuario actúa sobre ellos. Por ejemplo, cuando el usuario hace clic en un botón, se envía un evento que identifica al botón pulsado. Generalmente, el programa simplemente implementa la interfaz apropiada y registra un listener de eventos para cada control. Como se vio en el Capítulo 22, una vez instalado un listener, se le envían automáticamente los eventos. En las siguientes secciones, se especifica la interfaz apropiada para cada control.

La excepción de tipo HeadlessException

La mayoría de los controles AWT que se describen en este capítulo tienen constructores que pueden lanzar una excepción de tipo **HeadlessException** cuando se intenta hacer una instancia de un componente GUI en un ambiente no interactivo (tal como uno en el cual no hay un Mouse, o no hay un teclado presente). La excepción **HeadlessException** fue agregada por Java 1.4. Se puede utilizar esta excepción para escribir un código que pueda adaptarse a los ambientes no interactivos (aunque ciertamente, no siempre es posible.) Esta excepción no es manejada por los programas en este capítulo debido a que se requiere un ambiente interactivo para demostrar los controles de AWT.

Label

El control más sencillo de utilizar es la etiqueta. Una *etiqueta* es un objeto de la clase **Label**, y contiene una cadena que se muestra en la pantalla. Las etiquetas son controles pasivos que no admiten ninguna interacción con el usuario. Label define los siguientes constructores:

Label( ) throws HeadlessException
Label(String *str*) throws HeadlessException
Label(String *str*, int *alx*) throws HeadlessException

El primer constructor crea una etiqueta vacía. El segundo crea una etiqueta que contiene la cadena especificada en *str*. Este string está justificado a la izquierda. La tercera versión crea una etiqueta que contiene la cadena especificada por *str* utilizando el alineamiento especificado por *alx*. El valor de *alx* debe ser una de estas tres constantes: **Label.LEFT**, **Label.RIGHT**, o **Label.CENTER**.

Se puede establecer o cambiar el texto de una etiqueta utilizando el método **setText( ).** Se puede obtener el contenido de una etiqueta llamando al método **getText( ).** Estos métodos tienen el siguiente formato:

void setText(String *str*)
String getText( )

Para **setText( ),** *str* especifica la nueva etiqueta. Para **getText( ),** se devuelve la etiqueta actual.

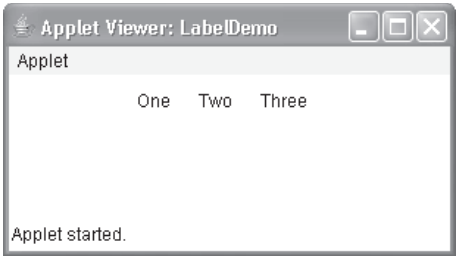
Se puede establecer el alineamiento del texto dentro de la etiqueta llamando al método **setAlignment( ).** Para obtener el alineamiento actual se llama a **getAlignment( ).** Estos métodos tienen el siguiente formato:

void setAlignment(int *alx*)
int getAlignment( )

Aquí, *alx* debe ser una de las constantes de alineamiento mencionadas anteriormente.

El siguiente ejemplo crea tres etiquetas y las añade a una ventana en un applet:

```
// Ejemplo con etiquetas
import java.awt.*;
import java.applet.*;
/*
<applet code="LabelDemo" width=300 height=200>
</applet>
*/

public class LabelDemo extends Applet {
  public void init() {
    Label primera = new Label("Uno");
    Label segunda = new Label("Dos");
    Label tercera = new Label("Tres");

    // añade las etiquetas a la ventana del applet
    add(primera);
    add(segunda);
    add(tercera);
  }
}
```



A continuación se muestra la ventana creada por el applet **LabelDemo**. Observe que las etiquetas están organizadas en la ventana utilizando el gestor de organización por omisión. Posteriormente, se verá cómo controlar de manera más precisa la posición de las etiquetas.

Button

Quizás el control que más se utiliza es el botón. Un *botón* es un componente que contiene una etiqueta y que genera un evento cuando es pulsado. Los *botones* son objetos de la clase **Button**. **Button** define dos constructores:

Button( ) throws HeadlessException
Button(String *str*) throws HeadlessException

El primer constructor crea un botón vacío, es decir, sin etiqueta. El segundo crea un botón que contiene *str* como etiqueta.

Una vez que se crea un botón, se le puede asignar una etiqueta llamando al método **setLabel( ).** También se puede obtener su etiqueta llamando al método **getLabel( ).** Estos métodos tienen el siguiente formato:

void setLabel(String *str*)
String getLabel( )

donde *str* especifica la nueva etiqueta del botón.

Gestión de botones

Cada vez que se pulsa un botón, se genera un evento de acción que se envía a cualquier listener que previamente haya registrado su interés por recibir información de eventos de acción generados por ese componente. Cada listener implementa de la interfaz **ActionListener**. Esta interfaz define al método **actionPerformed( )**, el cual es invocado cuando ocurre un evento. Como argumento a ese método se pasa un objeto **ActionEvent**. El objeto **ActionEvent** contiene una referencia al botón que ha generado el evento y una referencia a la cadena de comando asociada con el botón. Por omisión, la cadena de comando es la etiqueta del botón. Normalmente, se puede utilizar tanto la referencia al botón como la cadena de comando para identificar al botón (veremos ejemplos de esto más adelante).

El siguiente ejemplo crea tres botones con las etiquetas "Sí", "No", y "Sin decidir". Cada vez que se pulsa un botón, se muestra un mensaje que indica cuál botón se ha pulsado. En este ejemplo, se utiliza la etiqueta del botón para determinar qué botón se ha pulsado. La etiqueta se obtiene llamando al método **getActionCommand( )** sobre el objeto **ActionEvent** que el método **actionPerformed( )** recibe como parámetro.

```
// Ejemplo con botones
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="DemoBoton" width=250 height=150>
  </applet>
*/

public class DemoBoton extends Applet implements ActionListener {
  String msg = "";
  Button si, no, quiza;

  public void init() {
    si = new Button("Si");
```

```
    no = new Button("No");
    quiza = new Button("Sin decidir");

    add(si);
    add(no);
    add(quiza);

    si.addActionListener(this);
    no.addActionListener(this);
    quiza.addActionListener(this);
  }

  public void actionPerformed(ActionEvent ae) {
    String str = ae.getActionCommand();
    if (str.equals("Si")) {
      msg = "Has dado clic en Si.";
    }
    else if(str.equals("No"))
      msg = "Has dado clic en No.";
    }
    else {
      msg = "Has dado clic en Sin decidir.";
    }

    repaint();
  }

  public void paint(Graphics g) {
    g.drawString(msg, 6, 100);
  }
}
```

La salida del programa **DemoBoton** se muestra en la Figura 24-1.

Como se comentó antes, además de comparar las etiquetas de los botones, también se puede determinar qué botón se ha pulsado comparando el objeto obtenido con el método **getSource( )** con los objetos Button que se han añadido a la ventana. Para hacerlo, es necesario mantener una lista de los objetos que se van añadiendo a la ventana. El siguiente applet muestra esta solución:

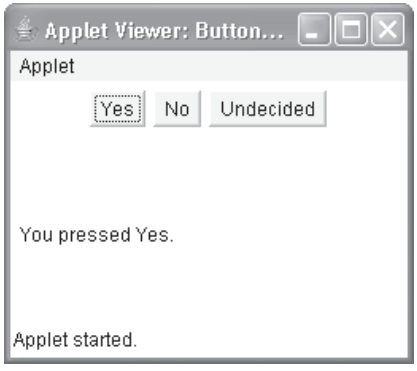


FIGURA 24-1 Ejemplo de salida del applet **DemoBoton**

```
// Reconocimiento de objetos de tipo Button.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="ListaBotones" width=250 height=150>
  </applet>
*/
pub1ic c1ass ListaBotones extends Applet imp1ements ActionListener {
  String msg = "";
  Button bList[] = new Button[3]:

  pub1ic void init() {
    Button si = new Button("Si");
    Button no = new Button("No");
    Button quiza = new Button{"Sin decidir");

    // se guardan las referencias a los botones conforme se añaden
    bList[0] = (Button) add(si);
    bList[l] = (Button) add(no);
    bList[2] = (Button) add(quiza);

    // se registran para recibir eventos de acción
    for (int i = 0; i < 3; i++) {
      bList[i].addActionListener(this);
    }
  }

  pub1ic void actionPerformed(ActionEvent ae) {
    for (int i = 0; i < 3; i++) {
      if (ae.getSource() = = bList[i]) {
        msg = "Ha presionado " + bList [i].getLabel( );
      }
    }
    repaint():
  }

  public void paint(Graphics g} {
    g.drawString(msg, 6, 100);
  }
}
```

En esta versión, el programa almacena cada referencia a un botón en un arreglo de botones cuando los botones se han añadido a la ventana del applet. Recordemos que el método **add( )** devuelve una referencia al mismo botón añadido. Dentro de **actionPerformed( )**, se utiliza este arreglo para determinar qué botón se ha presionado.

Para programas simples, suele ser más fácil reconocer los botones por sus etiquetas. Sin embargo, en situaciones en las que la etiqueta del botón cambia en tiempo de ejecución o en las que se utilizan varios botones con la misma etiqueta, puede ser más fácil determinar qué botón se ha presionando utilizando las referencias a los botones.

También es posible dar un valor a la cadena de comando, asociada con un botón, diferente a su etiqueta llamando al método **setActionCommand( )**. Este método cambia la cadena de

comando, pero no afecta a la cadena utilizada como etiqueta del botón. Definir una cadena de comando para el botón, nos permite diferenciarla de la etiqueta del botón.

Checkbox

Un *checkbox* es un control que se utiliza para activar o desactivar una opción. Está formado por un pequeño cuadro que puede contener o no una marca de comprobación. Hay una etiqueta asociada a cada **checkbox** que describe la opción representada. Para cambiar el estado de un **checkbox** sólo hay que dar un clic sobre él. Los **checkbox** se pueden utilizar individualmente o como parte de un grupo. Los **checkbox** son objetos de la clase **Checkbox**.

La clase **Checkbox** proporciona los siguientes constructores:

```
Checkbox( ) throws HeadlessException
Checkbox(String str) throws HeadlessException
Checkbox(String str, boolean on) throws HeadlessException
Checkbox(String str, boolean on, CheckboxGroup cbGroup) throws HeadlessException
Checkbox(String str, CheckboxGroup cbGroup, boolean on) throws HeadlessException
```

La primera forma crea un checkbox cuya etiqueta está inicialmente vacía y el estado del checkbox está como no seleccionado. El segundo formato crea un checkbox cuya etiqueta está especificada en *str* y el estado del checkbox está como no seleccionado. La tercera forma permite establecer el estado inicial del checkbox. Si *on* es **verdadero**, el checkbox está inicialmente seleccionado; en caso contrario estará como no seleccionado. Los formatos cuarto y quinto crean un checkbox cuya etiqueta viene especificada por *str* y cuyo grupo está especificado por *cbGroup*. Si este checkbox no pertenece a ningún grupo, *cbGroup* tiene que ser **null** (los grupos de checkbox se describen en la siguiente sección). El valor de *on* determina el estado inicial del checkbox.

Para obtener el estado actual de un checkbox, se llama al método **getState( )**. Para establecer el estado del checkbox se llama al método **setState( )**. Se puede obtener la etiqueta asociada a un checkbox llamando a **getLabel( )**. Llamando al método **setLabel( )** se establece una nueva etiqueta. Estos métodos tienen el siguiente formato:

```
boolean getState( )
void setState(boolean s)
String getLabel( )
void setLabel(String str)
```

Donde si *s* es **true**, el checkbox estará seleccionado. Si es **false**, el checkbox no estará seleccionado. La cadena que se pasa en *str* especifica la nueva etiqueta asociada al checkbox.

Gestión de checkbox

Cada vez que se selecciona o deselecciona un checkbox, se genera un evento que se envía a cualquier listener que previamente haya registrado su interés para recibir información de los eventos que se producen desde ese componente. Cada listener implementa la interfaz **ItemListener**. Esta interfaz define el método **itemStateChanged( )**. Como argumento este

método recibe un objeto **ItemEvent**, que encapsula la información sobre el evento (por ejemplo, si se ha seleccionado o deseleccionado).

El siguiente ejemplo crea cuatro checkbox. El estado inicial del primero es seleccionado. El programa muestra mediante un mensaje el estado de cada uno de los checkbox. Cada vez que cambia el estado de un checkbox, el programa actualiza los mensajes desplegados.

```
// Ejemplo con checkbox.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
  <applet code= "CheckboxDemo" width=250 height=200>
  </applet>
*/

public class CheckboxDemo extends Applet implements ItemListener {
  String msg = "";
  Checkbox WinXP, WinVista, solaris, mac;

  public void init( ) {
    WinXP = new Checkbox ("Windows XP", null, true);
    WinVista = new Checkbox("Windows Vista");
    solaris = new Checkbox("Solaris");
    mac = new Checkbox("MacOS");

    add(WinXP);
    add (WinVista);
    add(solaris);
    add(mac);

    WinXP.addItemListener(this);
    WinVista.addItemListener(this);
    solaris.addItemListener(this);
    mac.addItemListener(this);
  }

  public void itemStateChanged(ItemEvent ie) {
    repaint ( );
  }

  // muestra el estado actual de los checkbox
  public void paint(Graphics g) {
    msg = " Estado actual: ";
    g.drawString (msg, 6, 80);
    msg = " Windows XP: " + WinXP.getState();
    g.drawString(msg, 6, 100);
    msg = " Windows Vista: " + WinVista.getState();
    g.drawString(msg, 6, 120);
    msg = " Solaris: " + solaris.getState();
    g.drawString(msg, 6, 140);
    msg = " MacOS: " + mac.getState();
    g.drawString(msg, 6, 160);
  }
}
```

Un ejemplo de la salida de este programa se muestra en la Figura 24-2

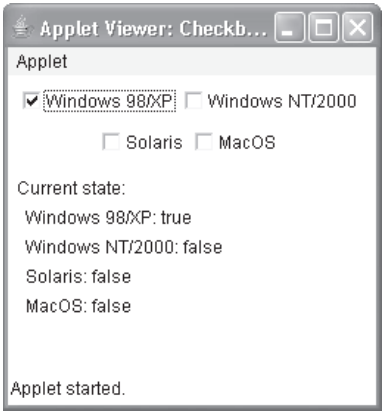


FIGURA 24-2 Ejemplo de salida del applet **CheckboxDemo**

CheckboxGroup

Es posible crear un conjunto de checkbox que sean mutuamente excluyentes en el que sólo se pueda elegir uno y sólo un checkbox del grupo. A estos checkbox se les suele llamar *botones de radio*, ya que funcionan como el selector de estaciones en la radio de un auto, donde sólo se puede seleccionar una estación a la vez. Para crear un conjunto de checkbox mutuamente excluyentes, primero hay que definir el grupo al que pertenecerán y después especificar ese grupo cuando se crean los checkbox. Los grupos de checkbox son objetos de la clase **CheckboxGroup**. La clase CheckboxGroup sólo define al constructor por omisión, el cual crea un grupo vacío.

Se puede determinar qué checkbox está actualmente seleccionado en un grupo llamando a **getSelectedCheckbox( )**. Se puede seleccionar un checkbox llamando al método **setSelectedCheckbox( )**. Estos métodos tienen las siguientes firmas:

Checkbox getSelectedCheckbox( )
void setSelectedCheckbox(Checkbox *cb*)

Donde *cb* es el checkbox que se quiere marcar como seleccionado. El checkbox que hasta entonces estaba seleccionado, en el grupo, se deseleccionará.

A continuación se presenta un programa que utiliza checkbox que son parte de un grupo:

```
// Ejemplo de grupos de checkbox
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
  <applet code= "CBGroup" width=250 height=200>
  </applet>
*/

public class CBGroup extends Applet implements ItemListener {
  String msg = "";
  Checkbox WinXP, WinVista, solaris, mac;
```

```
  CheckboxGroup cbg;

  public void init() {
    cbg = new CheckboxGroup( );
    WinXP = new Checkbox ("Windows XP", cbg, true);
    WinVista = new Checkbox("Windows Vista", cbg, false);
    solaris = new Checkbox("Solaris",cbg, false);
    mac = new Checkbox("MacOS", cbg, false);

    add(WinXP);
    add (WinVista);
    add(solaris);
    add(mac);

    WinXP.addItemListener(this);
    WinVista.addItemListener(this);
    solaris.addItemListener(this);
    mac.addItemListener(this);
  }

  public void itemStateChanged(ItemEvent ie) {
    repaint();
  }

  // Se muestra el estado actual de los checkbox.
  public void paint(Graphics g) {
    msg = "Estado actual: ";
    msg += cbg.getSelectedCheckbox( ).getLabel();
    g.drawString(msg, 6, 100);
  }
}
```

La salida generada por el applet **CBGroup** se muestra en la Figura 24-3. Note que los checkbox tienen forma circular.

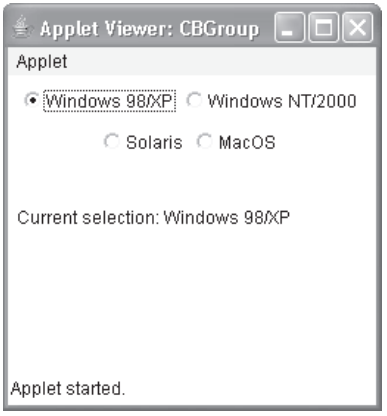


FIGURA 24-3 Ejemplo de salida del applet **CBGroup**

Choice

La clase **Choice** es utilizada para crear listas *desplegables* de elementos de los cuales el usuario podría escoger. De esta forma, un control de tipo **Choice** tiene forma de menú. Cuando está inactivo, un componente **Choice** ocupa sólo el espacio suficiente para mostrar al elemento actual seleccionado. Cuando el usuario hace clic en él aparece la lista completa de opciones, y se puede realizar una nueva selección. Cada elemento de la lista es una cadena que aparece justificada a la izquierda y en el orden en que se añadió al objeto **Choice**. **Choice** sólo define un constructor por omisión, que crea una lista vacía.

Para añadir una selección a la lista, hay que llamar al método **add( )**, que tiene los siguientes formatos:

```
void add(String nom)
```

Donde *nom* es el nombre del elemento que se añade. Los elementos se añaden a la lista en el orden en que se llama al método **add( )**.

Para determinar qué elemento está seleccionado actualmente, se llama al método **getSelectedItem( )** o al método **getSelectedIndex( )**. Estos métodos se muestran a continuación:

```
String getSelectedItem( )
int getSelectedIndex( )
```

El método **getSelectedItem( )** devuelve una cadena que contiene el nombre del elemento. El método **getSelectedIndex( )** devuelve la posición del elemento considerando que el primer elemento está en la posición 0. Por omisión, aparece seleccionado el primer elemento que se añadió a la lista.

Para obtener el número de elementos que hay en la lista, se llama al método **getItemCount( )**. Se puede definir al elemento que deseamos aparezca como seleccionado utilizando el método **select( )**, tanto con un entero que indique la posición del elemento (empezando a contar desde cero), como con una cadena que tenga el nombre que aparece en la lista. Estos métodos son los siguientes:

```
int getItemCount( )
void select(int index)
void select(String nom)
```

Dada una posición, se puede obtener el nombre del elemento que está en esa posición llamando al método **getItem( )**, que tiene el siguiente formato:

```
String getItem(int index)
```

Aquí, *index* especifica la posición del elemento deseado.

Gestión de Choice

Cada vez que se selecciona un elemento de tipo **Choice**, se genera un evento que se envía a todo listener que se haya registrado para recibir la información de los eventos de ese componente. Cada listener debe implementar la interfaz **ItemListener**, que define al método **itemStateChanged( )**, el cual recibe como argumento un objeto de la clase **ItemEvent**.

A continuación se muestra un ejemplo que crea dos menús de tipo **Choice**. Uno selecciona un sistema operativo. El otro selecciona a un navegador.

```
//Ejemplo del control Choice.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="ChoiceDemo" width=300 height=180>
  </applet>
*/

public class ChoiceDemo extends Applet implements ItemListener {
  Choice os, browser;
  String msg = "";

  public void init() {
    os = new Choice();
    browser = new Choice();

    // se añaden elementos a la lista del sistema operativo
    os.add("Windows XP");
    os.add("Windows Vista");
    os.add("Solaris");
    os.add("MacOS");

    // se añaden elementos a la lista del navegador
    browser.add("Internet Explorer");
    browser.add("Firefox");
    browser.select("Opera");

    // se añaden las listas a la ventana
    add(os);
    add(browser);

    // se registran para recibir eventos
    os.addItemListener(this);
    browser.addItemListener(this);
  }

  public void itemStateChanged(ItemEvent ie) {
    repaint();
  }

  // Se visualizan las selecciones actuales.
  public void paint(Graphics g) {
    msg = "Actual OS: ";
    msg += os.getSelectedItem();
    g.drawString(msg, 6, 120);
    msg = "Navegador actual: ";
    msg += browser.getSelectedItem();
    g.drawString(msg, 6, 140);
  }
}
```

La salida se muestra en la Figura 24-4.

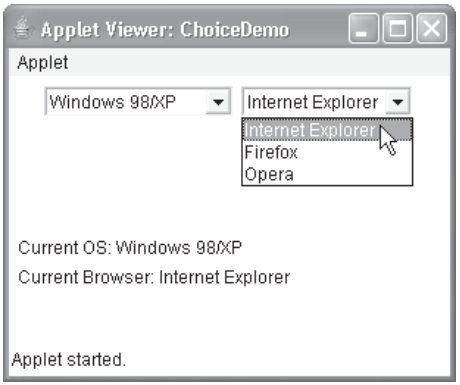


FIGURA 24-4 Ejemplo de salida del applet **ChoiceDemo**

List

La clase **List** proporciona una lista de selección compacta, con desplazamiento, que permite realizar selecciones múltiples. A diferencia del objeto **Choice**, que sólo muestra un único elemento que puede ser seleccionado en el menú, un objeto **List** puede ser construido para mostrar cualquier cantidad de opciones en una ventana. También se puede configurar de manera que sea posible realizar selecciones múltiples. La clase **List** tiene estos constructores:

List( ) throws HeadlessException
List(int *numRen*) throws HeadlessException
List(int *numRen*, boolean *multiple*) throws HeadlessException

La primera versión crea un objeto **List** que permite que haya sólo un elemento seleccionado en cada instante. En el segundo formato, el valor de *numRen* específica el número de entradas en la lista que estarán visibles (las demás se pueden ver cuando sea necesario moviendo la barra de desplazamiento del control). En el tercer formato, si "multiple" es verdadero, el usuario puede seleccionar dos o más elementos a la vez. Si es falso, sólo se puede seleccionar un elemento.

Para añadir un elemento a la lista, se llama al método **add( )**, que tiene estos dos formatos:

void add(String *nom*)
void add(String *nom*, int *index*)

Donde *nom* es el nombre del elemento que se añade a la lista. El primer formato añade elementos al final de la lista. El segundo formato los añade en la posición indicada en *index*, que comienza a contar desde cero. Se puede utilizar el valor –1 si se quiere añadir el elemento al final de la lista.

En las listas que sólo permiten seleccionar un único elemento, se puede determinar qué elemento está actualmente seleccionado llamando tanto a **getSelectedItem( )** como a **getSelectedIndex( )**. Estos métodos se muestran a continuación:

```
String getSelectedItem( )
int getSelectedIndex( )
```

El método **getSelectedItem( )** devuelve una cadena que contiene el nombre del elemento. Si se selecciona más de un elemento o si no se selecciona ninguno, se devuelve **null**.

En las listas que permiten realizar una selección múltiple, los métodos **getSelectedItems( )** o **getSelectedIndexes( )**, se utilizan para determinar qué elementos están seleccionados actualmente:

```
String[] getSelectedItems( )
int[] getSelectedIndexes( )
```

getSelectedItems( ) devuelve un arreglo con los nombres de los elementos seleccionados actualmente. **getSelectedIndexes( )** devuelve un arreglo con las posiciones de los elementos seleccionados actualmente.

Para obtener el número de elementos que hay en la lista, se llama al método **getItemCount( )**. Se puede establecer el elemento que tiene que estar seleccionado utilizando el método **select( )**, pasándole un entero que indique la posición del elemento que se desea seleccionar (empezando a contar por cero). Estos métodos tienen las siguientes firmas:

```
int getItemCount( )
void select(int index)
```

Dada una posición, se puede obtener el nombre del elemento que está en esa posición llamando al método **getItem( )**, el cual tiene el siguiente formato:

```
String getItem(int index)
```

Aquí, *index* especifica la posición del elemento deseado.

Gestión de List

Para procesar eventos de listas, es necesario implementar la interfaz **ActionListener**. Cada vez que se hace doble clic en un elemento List, se crea o genera un objeto de la clase **ActionEvent**. Se puede utilizar al método **getActionCommand( )** de la clase **ActionEvent** para conocer el nombre del elemento recién seleccionado. Cada vez que se selecciona o deselecciona un elemento con un solo clic, se genera un objeto de la clase **ItemEvent**. Se puede utilizar al método **getStateChange( )** de la clase **ItemEvent** para determinar si una selección o una deselección ha desencadenado este evento. El método **getItemSelectable( )** devuelve una referencia al objeto que ha desencadenado este evento.

A continuación se presenta un ejemplo que convierte los controles **Choice** del apartado anterior en componentes **List**, uno de opción múltiple y el otro de opción única:

```
// Ejemplo del componente List
import java.awt.*;
import java.awt.event.*;
```

```
import java.applet.*;
/*
  <applet code="ListDemo" width=300 height=180>
  </applet>
*/

public class ListDemo extends Applet implements ActionListener {
  List so, navegador;
  String msg = "";

  public void init() {
    so = new List(4, true);
    navegador = new List(4, false);

    // se añaden elementos a la lista de sistemas operativos
    so.add("Windows XP");
    so.add("Windows Vista");
    so.add("Solaris");
    so.add("MacOS");

    // se añaden elementos a la lista del navegador
    navegador.add("Internet Explorer");
    navegador.add("Firefox");
    navegador.add("Opera");
    navegador.select(1);

    // se añaden las listas a la ventana
    add(so);
    add(navegador);

    // se registran para recibir eventos de acción
    so.addActionListener(this);
    navegador.addActionListener(this);
  }

  public void actionPerformed(ActionEvent ae) {
    repaint();
  }

  // muestra las selecciones actuales.
  public void paint(Graphics g) {
    int idx[];

    msg = "Sistema operativo actual: ";
    idx = so.getSelectedIndexes();
    for(int i=0; i<idx.length; i++)
      msg += so.getItem(idx[i]) + " ";
    g.drawString (msg, 6, 120);
    msg = "Navegador actual: ";
    msg += navegador.getSelectedItem( );
    g.drawString(msg, 6, 140);
  }
}
```

La salida generada por el applet **ListDemo** se muestra en la Figura 24-5.

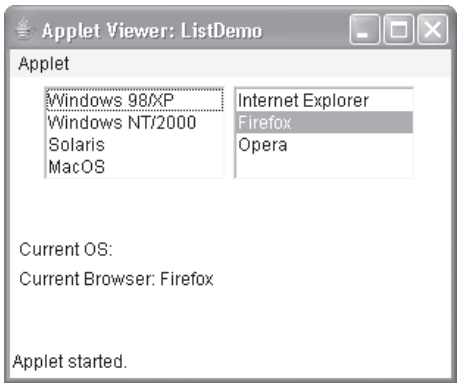


FIGURA 24-5 Ejemplo de salida del applet **ListDemo**

Scrollbar

Las *barras de desplazamiento* (*scrollbar*) se utilizan para seleccionar valores continuos localizados entre un valor mínimo y un valor máximo especificados. Las barras de desplazamiento pueden ser horizontales o verticales. Una barra de desplazamiento tiene varios elementos. Cada extremo de la barra tiene una flecha en la que se puede hacer clic para mover una unidad en la dirección de la flecha el valor actual de la barra de desplazamiento. El valor actual de la barra de desplazamiento, en relación a sus valores mínimo y máximo está marcado por un cuadrado (o caja de deslizamiento), que el usuario puede arrastrar hasta una nueva posición y la barra reflejará este valor. El usuario puede hacer clic en la zona de la barra de desplazamiento en que no está el cuadrado deslizante, para que éste salte a esa posición logrando incrementos mayores a 1. Esta acción se puede convertir en una forma de avance de página o retroceso de página. La clase **Scrollbar** modela al elemento barra de desplazamiento.

La clase **Scrollbar** define los siguientes constructores:

Scrollbar( ) throws HeadlessException
Scrollbar(int *estilo*) throws HeadlessException
Scrollbar(int *estilo*, int *inicio*, int *tamaño*, int *min*, int *max*) throws HeadlessException

El primer formato crea una barra de desplazamiento vertical. El segundo y tercer formatos permiten determinar la orientación de la barra de desplazamiento. Si *estilo* es **Scrollbar.VERTICAL**, se crea una barra de desplazamiento vertical. Si *estilo* es **Scrollbar.HORIZONTAL**, la barra de desplazamiento es horizontal. En el tercer formato, el *valor inicial* de la barra de desplazamiento se pasa con inicio. El número de unidades que representa la altura del cuadrado deslizante es *tamaño*.Y los valores mínimo y máximo de la barra de desplazamiento se especifican con *min* y *max*.

Si se construye una barra de desplazamiento utilizando uno de los dos primeros constructores, será necesario establecer luego sus parámetros utilizando el método **setValues( )**, mostrado aquí, antes de que pueda ser utilizada la barra:

void setValues(int *inicio*, int *tamaño*, int *min*, int *max*)

Los parámetros tienen el mismo significado que en el tercer constructor ya comentado.

Para obtener el valor actual de la barra de desplazamiento, se llama al método **getValue( )**. Para establecer el valor actual, se puede llamar al método **setValue( )**. Estos métodos tienen las siguientes firmas:

```
int getValue( )
void setValue(int valor)
```

Donde *valor* especifica el nuevo valor de la barra de desplazamiento. Cuando se establece un valor, el cuadrado de la barra de desplazamiento se mueve hasta la nueva posición.

También se pueden obtener los valores mínimo y máximo llamando a los métodos **getMinimum( )** y **getMaximum( )**:

```
int getMinimum( )
int getMaximum( )
```

Estos métodos devuelven la cantidad solicitada.

Por omisión, el incremento que se suma o resta a la barra de desplazamiento cada vez que se hace clic en la flecha es 1, pero se puede cambiar invocando al método **setUnitIncrement( )**. Por omisión, los incrementos de avance y retroceso de página tienen un valor de 10, y se pueden cambiar utilizando el método **setBlockIncrement( )**. Estos métodos se muestran a continuación:

```
void setUnitIncrement(int valor)
void setBlockIncrement(int valor)
```

Gestión de Scrollbar

Para procesar los eventos de barras de desplazamiento, se necesita implementar la interfaz **AdjustmentListener**. Cada vez que un usuario interactúa con una barra de desplazamiento, se genera un objeto de la clase **AdjustmentEvent**. Se puede utilizar su método **getAdjustment Type( )** para determinar el tipo de ajuste realizado. Los tipos posibles de ajuste son los siguientes:

BLOCK_DECREMENT	Se ha generado un evento de retroceso de página.
BLOCK_INCREMENT	Se ha generado un evento de avance de página.
TRACK	Se ha generado un evento de movimiento del cuadrado.
UNIT_DECREMENT	Se ha pulsado la flecha de la barra de desplazamiento hacia abajo.
UNIT_INCREMENT	Se ha pulsado la flecha de la barra de desplazamiento hacia arriba.

El siguiente ejemplo crea dos barras de desplazamiento, una horizontal y otra vertical y visualiza los valores actuales de esas barras de desplazamiento. Si se arrastra el ratón dentro de la ventana, se utilizan las coordenadas del ratón para actualizar las barras de desplazamiento y se visualiza un asterisco en la posición del ratón.

```
// Ejemplo de barras de desplazamiento.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="SBDemo" width=300 height=200>
  </applet>
*/
```

```
public class SBDemo extends Applet
  implements AdjustmentListener, MouseMotionListener {
  String msg = "";
  Scrollbar vertSB, horzSB;

  public void init() {
    int width = Integer.parseInt(getParameter("width"));
    int height = Integer.parseInt(getParameter("height"));

    vertSB = new Scrollbar(Scrollbar.VERTICAL,
                           0, 1, 0, height);
    horzSB = new Scrollbar(Scrollbar.HORIZONTAL,
                           0, 1, 0, width);
    add(vertSB);
    add(horzSB);

    // se registran para recibir eventos de ajuste
    vertSB.addAdjustmentListener(this);
    horzSB.addAdjustmentListener(this);

    addMouseMotionListener(this);
  }

  public void adjustmentValueChanged(AdjustmentEvent ae) {
    repaint();
  }

  // Se actualizan las barras de desplazamiento para mostrar el arrastre del
  ratón.
    public void mouseDragged(MouseEvent me) {
    int x = me.getX();
    int y = me.getY();
    vertSB.setValue(y);
    horzSB.setValue(x);
    repaint();
  }

  // Necesario para MouseMotionListener
  public void mouseMoved(MouseEvent me) {
  }

  // Se muestran los valores actuales de las barras de desplazamiento.
  public void paint(Graphics g) {
    msg = "Vertical: " + vertSB.getValue();
    msg += ", Horizontal: " + horzSB.getValue();
    g.drawString (msg, 6, 160);

    // se muestra la posición actual del ratón
    g.drawString("*", horzSB.getValue( ),
                 vertSB.getValue( ));
  }
}
```

La salida del applet **SBDemo** se muestra en la Figura 24-6

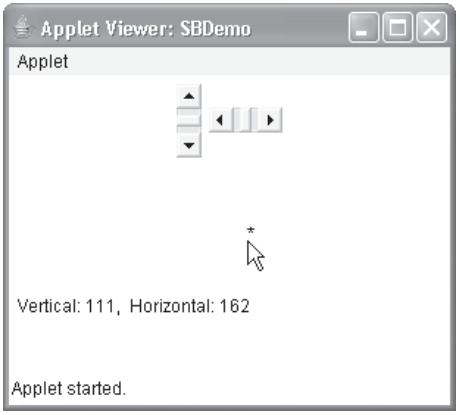


FIGURA 24-6 Ejemplo de salida del applet **SBDemo**

TextField

La clase **TextField** implementa un campo de texto de una sola línea, usualmente llamado control de edición. Los campos de texto permiten al usuario introducir cadenas y editar texto utilizando los cursores, las teclas de cortar y pegar, y las selecciones que se hacen con el ratón. La clase **TextField** es una subclase de **TextComponent**. **TextField** proporciona los siguientes constructores:

TextField( ) throws HeadlessException
TextField(int *numChars*) throws HeadlessException
TextField(String *str*) throws HeadlessException
TextField(String *str*, int *numChars*) throws HeadlessException

La primera versión crea un campo de texto por omisión. El segundo formato crea un campo de texto que tiene un tamaño de *numChars* caracteres. La tercera versión inicializa el campo de texto con la cadena dada mediante *str*. El cuarto formato inicializa un campo de texto con la cadena *str* y además especifica su tamaño.

La clase **TextField** (y su superclase **TextComponent**) proporciona varios métodos que permiten utilizar al campo de texto. Para obtener la cadena del campo de texto se llama al método **getText( ).** Para establecer un determinado texto se utiliza el método **setText( )**. Estos métodos tienen los siguientes formatos:

String getText( )
void setText(String *str*)

donde *str* es la cadena que se introduce en el campo de texto.

El usuario puede seleccionar una parte del texto del campo de texto utilizando el método **select( )**. Puede obtenerse el texto actualmente seleccionado llamando al método **getSelectedText( )**. A continuación se muestran estos métodos:

String getSelectedText( )
void select(int *startIndex*, int *endIndex*)

getSelectedText( ) devuelve el texto seleccionado. El método **select( )** selecciona los caracteres que comienzan en **startIndex** y terminan en *endIndex*-l.

Podemos controlar si el contenido de un campo de texto puede ser modificado por el usuario llamando al método **setEditable( )**. Puede determinarse si un campo de texto es editable o no llamando a **isEditable( )**. Estos métodos tienen los siguientes formatos:

```
boolean isEditable( )
void setEditable(boolean editable)
```

isEditable( ) devuelve **verdadero** si se puede cambiar el texto y **falso** en caso contrario. En **setEditable( )**, si el parámetro editable es verdadero, el texto podrá cambiar, y si es falso, el texto no se podrá cambiar.

Habrá ocasiones en que nos interese que el texto que introduce el usuario no sea visible, como cuando se introduce una clave de acceso. Se puede evitar que se muestre el texto que se escribe en el campo de texto llamando al método **setEchoChar( )**. Este método específica al carácter que el **TextField** desplegará en lugar de los caracteres escritos por el usuario (de esta forma, los caracteres reales no serán mostrados). Se puede revisar si campo de texto está en el modo de ocultar caracteres llamando al método **echoCharIsSet( )** y se puede obtener el carácter establecido como remplazo de los caracteres escritos llamando al método **getEchoChar( )**. A continuación se muestran esos métodos:

```
void setEchoChar(char ch)
boolean echoCharIsSet( )
char getEchoChar( )
```

Donde, *ch* especifica al carácter que se va a mostrar en el campo de texto.

Gestión de TextField

Dado que los campos de texto gestionan sus propias funciones de edición, generalmente el programa no tendrá que ocuparse de los eventos de teclas que ocurran dentro de un campo de texto. Sin embargo, puede ser que se desee responder cuando el usuario pulse la tecla ENTER; cuando el usuario pulsa ENTER se genera un evento de acción.

A continuación se muestra un ejemplo que crea la clásica ventana que pide el nombre y clave del usuario:

```
// Ejemplo con campos de texto.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="TextFieldDemo" width=380 height=l50>
  </applet>
*/

public class TextFieldDemo extends Applet
  implements ActionListener {

  TextField name, pass;

  public void init() {
    Label namep = new Label ("Nombre: ", Label. RIGHT);
    Label passp = new Label ("password: ", Label.RIGHT);
```

```
    name = new TextField(12);
    pass = new TextField(8);
    pass.setEchoChar('?');
    add(namep);
    add(name);
    add(passp);
    add(pass);

    // se registran para recibir los eventos de acción
    name.addActionListener(this);
    pass.addActionListener(this);
  }

  // El usuario presiona Enter.
  public void actionPerformed(ActionEvent ae) {
    repaint();
  }

  public void paint(Graphics g) {
    g.drawString("Nombre: " + name.getText(), 6, 60);
    g.drawString("Texto seleccionado en Nombre: "
                 + name.getSelectedText(), 6, 80);
    g.drawString("Clave: " + pass.getText(), 6, 100);
  }
}
```

La salida del applet **TextFieldDemo** se muestra en la Figura 24-7

TextArea

Algunas veces no basta con una entrada de una sola línea para realizar ciertas tareas. Para esas situaciones, AWT incluye un sencillo editor multilíneas llamado **TextArea**. Los constructores de **TextArea** son los siguientes:

TextArea( ) throws HeadlessException
TextArea(int *numLines*, int *numChars*) throws HeadlessException
TextArea(String *str*) throws HeadlessException
TextArea(String *str*, int *numLines*, int *numChars*) throws HeadlessException
TextArea(String *str*, int *numLines*, int *numChars*, int *sBars*) throws HeadlessException

FIGURA 24-7 Ejemplo de salida del applet **TextFieldDemo**

Aquí, *numLines* especifica la altura, en líneas, del área de texto, y *numChars* especifica su anchura en caracteres. Se puede especificar un texto inicial con *str*. En el quinto formato se pueden especificar las barras de desplazamiento que va a tener el control mediante sBars, *sBars* debe tomar uno de los siguientes valores:

SCROLLBARS_BOTH	SCROLLBARS_NONE
SCROLLBARS_HORIZONTAL_ONLY	SCROLLBARS_VERTICAL_ONLY

La clase **TextArea** es una subclase de **TextComponent**. Por tanto, soporta los métodos **getText( )**, **setText( )**, **getSelectedText( )**, **select( ), isEditable( ), y setEditable( )** descritos en la sección anterior.

TextArea añade los siguientes métodos:

void append(String *str*)
void insert(String *str*, int *index*)
void replaceRange(String *str*, int *startlndex*, int *endlndex*)

El método **append( )** añade la cadena especificada en *str* al final del texto actual. El método **insert( )** inserta la cadena que se pasa en *str* en el punto especificado en index. Para reemplazar texto hay que llamar al método **replaceRange( )**, que reemplaza los caracteres desde **startIndex** hasta **endIndex**–l con el texto dado en *str*.

Las áreas de texto son controles casi autocontenidos, por lo que el programa no puede entrar en su gestión interna. Las áreas de texto sólo generan eventos de obtención y pérdida del foco. Normalmente el programa simplemente obtiene el texto escrito en el componente cuando lo necesita.

El siguiente programa crea un control de tipo **TextArea**:

```
// Ejemplo de TextArea.
import java.awt.*:
import java.applet.*;
/*
<applet code= "TextAreaDemo" width=300 height=250>
</applet>
*/

public class TextAreaDemo extends Applet {
  public void init() {
    String val =
     "Java SE 6 es la última versión del lenguaje de programación más\n" +
     "ampliamente utilizado para programar en Internet.\n" +
     "Sobre la base de un rico patrimonio, Java ha desarrollado tanto\n" +
     "el arte como la ciencia detrás del diseño de lenguajes computacionales.\n" +
     "Una de las razones por las que Java es un continuo éxito, es debido a\n" +
     "su constante y firme evolución. Java nunca ha parado.\n" +
     "En lugar de eso, Java se ha adaptado constantemente a los\n" +
     "rápidos cambios en el panorama del mundo de las redes.\n" +
     "Más aún, Java ha conducido con frecuencia las formas, trazando\n" +
     "el camino para ser seguido por otros."

    TextArea text = new TextArea(val, 10, 30);
    add(text);
  }
}
```

La salida del applet **TextAreaDemo** es la siguiente:

Gestores de organización

Todos los componentes que se han mostrado en lo que va del capítulo han sido situados en sus ventanas utilizando el gestor de organización por omisión. Como se mencionó al inicio del capítulo, un gestor de organización coloca automáticamente los controles dentro de una ventana utilizando algún tipo de algoritmo. Si se ha programado en otros entornos gráficos, como Windows, se estará acostumbrado a colocar los controles manualmente. Aunque en Java también permite colocar manualmente los componentes en la ventana, generalmente esto no se hace por dos razones. Primero, porque es muy tedioso colocar a mano un número elevado de componentes. Segundo, porque a veces, cuando se va a colocar un control, todavía no se sabe su anchura y altura, ya que aún no se han diseñado todos los componentes. Esto es similar al dilema del huevo y la gallina; es muy confuso determinar cuando es el mejor momento para utilizar el tamaño de los componentes para colocarlo en una posición relativa a otros componentes.

Cada objeto de tipo **Container** tiene un gestor de organización asociado a él. Un gestor de organización es una instancia de cualquier clase que implementa la interfaz **LayoutManager**. El gestor de organización se establece mediante el método **setLayout( )**. Si no se llama al método **setLayout( )**, se utiliza el gestor de organización por omisión. Siempre que se cambie el tamaño de un contenedor (o que se establezca su tamaño por primera vez), se utiliza el gestor de organización para reposicionar todos los componentes que contiene.

El método **setLayout( )** tiene el siguiente formato:

void setLayout(LayoutManager *layoutObj*)

Donde *layoutObj* es una referencia al gestor de organización deseado. Si no se quiere utilizar ningún gestor de organización y se desea colocar los componentes manualmente, *layoutObj* debe ser **null**. Si se hace esto, habrá que determinar la forma y la posición de cada componente manualmente utilizando el método **setBounds( )** definido por **Component**. Pero lo habitual es utilizar un gestor de organización.

Cada gestor de organización mantiene una lista con los nombres de los componentes que almacena. Cada vez que se añade un componente a un contenedor, se informa de ello al gestor

de organización. Siempre que haya que cambiar el tamaño del contenedor, se consulta al gestor de organización por medio de los métodos **minimumLayoutSize( )** y **preferredLayoutSize( )**. Los componentes que se gestionan con un gestor de organización contienen los métodos **getPreferredSize( )** y **getMinimumSize( )**, que devuelven el tamaño más adecuado y el tamaño mínimo para visualizar el componente. Siempre que sea posible, el gestor de organización utilizará esos valores, intentando mantener la integridad de la organización. Estos métodos pueden ser sobrescritos en subclases que representan a controles propios. Si no se sobrescriben, se utilizan los valores por omisión.

Java tiene predefinidas varias clases **LayoutManager**, algunas de ellas se describen a continuación. El programador utiliza el gestor de organización que mejor se adecue a la aplicación que esté desarrollando.

FlowLayout

FlowLayout es el gestor de organización por omisión, y el que se ha utilizado en los ejemplos anteriores. **FlowLayout** implementa un estilo de organización que es parecido a como se sitúan las palabras en un editor de textos: una detrás de otra. La dirección en que se colocan los componentes, la controla el contenedor con su propiedad de orientación, la cual por omisión es, de izquierda a derecha y de arriba abajo. Cuando no caben más componentes en una línea, el siguiente se coloca en la siguiente línea. Entre componente y componente se deja un pequeño espacio, tanto arriba y abajo como a izquierda y derecha. A continuación se muestran los constructores de **FlowLayout**:

FlowLayout( )
FlowLayout(int *how*)
FlowLayout(int *how*, int *horz*, int *vert*)

Con la primera forma se realiza la organización por omisión, que centra los componentes y deja cinco píxeles entre componente y componente. Con la segunda forma se puede especificar cómo se organiza cada línea. Los valores que se pueden dar a *how* son los siguientes:

FlowLayout.LEFT
FlowLayout.CENTER
FlowLayout.RIGHT
FlowLayout.LEADING
FlowLayout.TRAILING

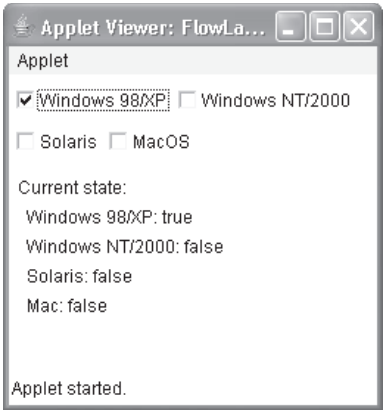
Estos valores especifican una alineación a la izquierda, centrada, a la derecha, borde delantero, borde trasero respectivamente. El tercer formato permite especificar el espacio horizontal y vertical que se deja entre los componentes mediante *horz* y *vert*, respectivamente.

A continuación se presenta una versión del applet **CheckboxDemo**, que ha aparecido antes en este capítulo, modificada para que tenga una organización alineada a la izquierda.

```
// FlowLayout con alineación a la izquierda.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="FlowLayoutDemo" width=250 height=200>
  </applet>
*/
```

```
public class FlowLayoutDemo extends Applet
  implements ItemListener {

  String msg = "";
  Checkbox WinXP, WinVista, solaris, mac;

  public void init() {
    // se establece la alineación a la izquierda
    setLayout(new FlowLayout(FlowLayout.LEFT));

    WinXP = new Checkbox("Windows XP", null, true);
    WinVista = new Checkbox("Windows Vista");
    solaris = new Checkbox("Solaris");
    mac = new Checkbox("MacOS");

    add(WinXP);
    add(WinVista);
    add(solaris);
    add (mac);

    // se registran para recibir eventos
    WinXP.addItemListener(this);
    WinVista.addItemListener(this);
    solaris.addItemListener(this);
    mac.addItemListener(this);
  }

  // Se vuelve a pintar cuando cambia el estado de un checkbox.
  public void itemStateChanged(ItemEvent ie) {
  repaint();
  }

  // Muestra el estado actual de los checkbox.
  public void paint(Graphics g) {

    msg = "Estado actual: ";
    g.drawString(msg, 6, 80);
    msg = " Windows XP: " + WinXP.getState();
    g.drawString (msg, 6, 100);
    msg = " Windows Vista: " + WinVista.getState();
    g.drawString(msg, 6, 120);
    msg = " Solaris: " + solaris.getState();
    g.drawString(msg, 6, 140);
    msg = " Mac: " + mac.getState();
    g.drawString(msg, 6, 160);
  }
}
```

La salida generada por el applet **FlowLayoutDemo** es la siguiente. Compare esta salida con la del applet **CheckboxDemo** que aparece en la Figura 24-2.



BorderLayout

La clase **BorderLayout** implementa un estilo de organización habitual para ventanas de nivel superior. Tiene cuatro componentes de anchura fija en los bordes y un área grande

en el centro. Los cuatro lados son identificados con los nombres north (norte), south (sur), east (este) y west(oeste); y al área central se le llama center (centro). Los constructores definidos por **BorderLayout** son los siguientes:

BorderLayout( )
BorderLayout(int *horz*, int *vert*)

El primer constructor crea una organización por omisión. El segundo permite especificar el espacio horizontal y vertical que se deja entre los componentes mediante *horz* y *vert*, respectivamente.

BorderLayout define las siguientes constantes que especifican las regiones:

BorderLayout.CENTER	BorderLayout.SOUTH
BorderLayout.EAST	BorderLayout.WEST
BorderLayout.NORTH	

Cuando se añaden componentes, se debe utilizar estas constantes con el siguiente formato del método **add( )**:

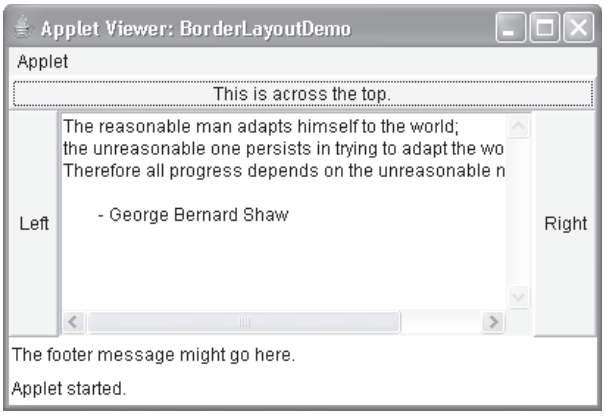
void add(Component *compObj*, Object *region*);

Donde, *compObj* es el componente que se va a añadir, y *región* especifica dónde se va a añadir.

A continuación se muestra un ejemplo de un **BorderLayout** con un componente en cada área:

```
// Ejemplo de BorderLayout.
import java.awt.*;
import java.applet.*;
import java.util.*;
/*
<applet code="BorderLayoutDemo" width=400 height=200>
</applet>
*/

public class BorderLayoutDemo extends Applet {
  public void init() {
    setLayout(new BorderLayout());

    add(new Button("Esto está en la parte superior."),
        BorderLayout.NORTH);
    add(new Label("El mensaje de pie de página podría ir aquí."),
        BorderLayout.SOUTH);
    add(new Button("Derecha"), BorderLayout.EAST);
    add(new Button("Izquierda") , BorderLayout.WEST):

    String msg = "El hombre razonable se adapta " +
      "al mundo;\n" +
      "el hombre no razonable intenta que " +
      "el mundo se adapte a él.\n" +
      "Por tanto todo el progreso depende " +
      "del hombre no razonable.\n\n" +
      "        - George Bernard Shaw\n\n";

    add(new TextArea(msg), BorderLayout.CENTER);
  }
}
```

La salida del applet **BorderLayoutDemo** se muestra a continuación:



Insets

Algunas veces será deseable dejar un pequeño espacio entre el contenedor donde están los componentes y la ventana que lo contiene. Para hacerlo, hay que sobrescribir el método **getInsets( )**, que está definido por la clase **Container**. Este método devuelve un objeto **Insets** que contiene las separaciones superior, inferior, izquierda y derecha que se van a utilizar cuando se visualice el contenedor. El gestor de organización utiliza estos valores para colocar los componentes al organizar la ventana. El constructor de **Insets** es el siguiente:

Insets(int *top*, int *left*, int *bottom*, int *right*)

Los valores que se pasan en *top*, *left*, *bottom* y *right* especifican el espacio que tiene que haber entre el contenedor y el borde de la ventana.

El método **getInsets( )** tiene el siguiente formato:

Insets getInsets( )

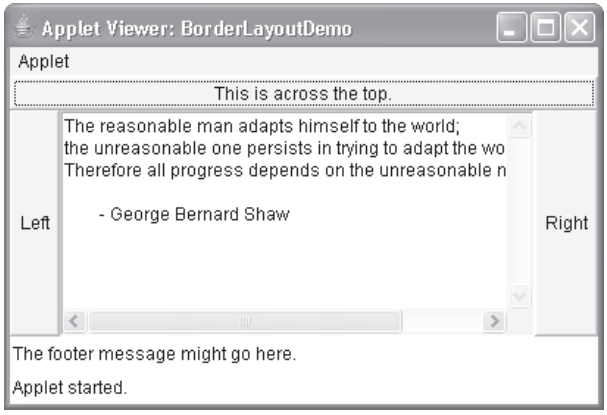
Cuando se sobrescribe uno de estos métodos, se devuelve un nuevo objeto **Insets** que contiene los márgenes deseados.

A continuación se presenta el ejemplo anterior de **BorderLayout** modificado para que sus componentes estén separados diez píxeles de cada borde. Se ha establecido como color de fondo el cyan para apreciar mejor el efecto.

```
// Ejemplo de BorderLayout con márgenes.
import java.awt.*;
import java.applet.*;
import java.util.*;
/*
<applet code="InsetsDemo" width=400 height=200>
</applet>
*/

public class InsetsDemo extends Applet {
  public void init() {
    // se establece el color de fondo para que se vean fácilmente los márgenes
```

```
    setBackground(Color.cyan);

    setLayout(new BorderLayout());

    add(new Button("Esto está en la parte superior."),
      BorderLayout.NORTH);
    add(new Label("El mensaje de pie de página podría ir aquí."),
      BorderLayout.SOUTH);
    add(new Button("Derecha"), BorderLayout.EAST);
    add(new Button("Izquierda"), BorderLayout.WEST);

    String msg = "El hombre razonable se adapta " +
      "al mundo; \n" +
      "el hombre no razonable intenta que " +
      "el mundo se adapte a él. \n" +
      "Por tanto todo el progreso depende " +
      "del hombre no razonable.\n\n" +
      " - George Bernard Shaw\n\n";

    add(new TextArea(msg), BorderLayout.CENTER),
  }
  // añade insets
  public Insets getInsets() {
    return new Insets(10, 10, 10, 10);
  }
}
```

La salida del applet **InsetsDemo** es la siguiente:



GridLayout

GridLayout organiza los componentes en una cuadrícula de dos dimensiones. Cuando se crea una instancia de **GridLayout**, se deben establecer el número de filas y de columnas. Los constructores de GridLayout son los siguientes:

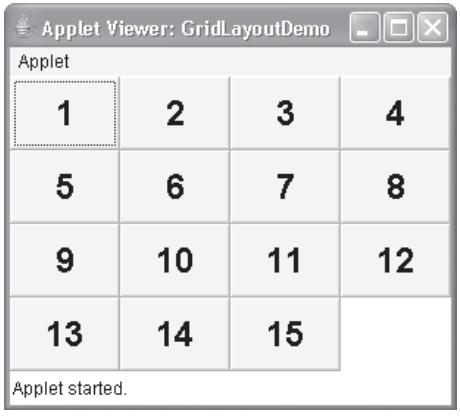
GridLayout( )
GridLayout(int *numF*, int *numC*)
GridLayout(int *numF*, int *numC*, int *horz*, int *vert*)

La primera forma crea una cuadrícula de una sola columna. La segunda forma crea una cuadrícula con *numF* filas y *numC* columnas. La tercera forma permite especificar el espacio horizontal y vertical que se deja entre los componentes mediante *horz* y *vert*, respectivamente. Tanto *numF* como *numC* pueden ser cero; si se iguala *numF* a cero, las columnas son de longitud ilimitada, y si se iguala *numC* a cero, las filas son de longitud ilimitada.

A continuación se presenta un programa que crea una cuadrícula de 4x4 y la rellena con 15 botones, cada uno etiquetado con su índice:

```
// Ejemplo de GridLayout
import java.awt.*;
import java.applet.*;
/*
<applet code="GridLayoutDemo" width=300 height=200>
</applet>
*/

public class GridLayoutDemo extends Applet {
  static final int n = 4;
  public void init() {
    setLayout(new GridLayout(n, n));

  setFont(new Font("SansSerif", Font.BOLD, 24));

  for(int i = 0; i < n; i++) {
    for(int j = 0; j < n; j++) {
      int k = i * n + j;
      if(k > 0)
        add(new Button("" + k));
      }
    }
  }
}
```

La salida generada por el applet **GridLayoutDemo** se muestra a continuación:



NOTA *Este programa puede ser el principio para desarrollar un rompecabezas de 15 cuadros.*

CardLayout

La clase **CardLayout** es distinta a los demás gestores de organización ya que almacena diferentes niveles. Cada nivel puede verse como si fuese una carta o tarjeta con un número que se puede barajar de tal manera que en cada momento puede haber una carta encima de las demás. Esto puede ser útil para interfaces de usuario con componentes opcionales que se puedan habilitar y deshabilitar dinámicamente en función de la entrada del usuario. Se pueden crear los diferentes niveles y después ocultarlos, dejándolos preparados para cuando se necesiten.

CardLayout tiene estos dos constructores:

CardLayout( )
CardLayout(int *horz*, int *vert*)

La primera forma crea al gestor por omisión. La segunda forma permite especificar el espacio horizontal y vertical que se deja entre los componentes mediante *horz* y *vert*, respectivamente.

Utilizar este gestor lleva un poco más de trabajo que el resto de los gestores. Las cartas se almacenan normalmente en un objeto de la clase **Panel**. Este panel debe tener seleccionado el gestor **CardLayout** como su gestor de organización. Las cartas también suelen ser objetos de la clase **Panel**. Se debe crear un panel que contenga todas las cartas, además de un panel para cada una de las cartas. Después, se añaden los componentes que forman la carta o tarjeta al panel apropiado. Luego se añaden esos paneles al panel que tiene a **CardLayout** como gestor de organización. Finalmente, hay que añadir este último panel al panel principal del applet. Una vez hecho todo esto, hay que dar al usuario una manera de poder elegir una de las cartas. Una forma suele ser incluir un botón para cada carta o tarjeta.

Cuando se añaden las cartas al panel principal, se les suele dar un nombre. Normalmente se utiliza el método **add( )** para añadir una carta a un panel como sigue:

void add(Component *panelObj*, Object *nom*);

Donde *nom* es una cadena que especifica el nombre de la carta o tarjeta del panel dado en *panelObj.*

Después de haber creado todas las cartas, el programa puede activar una de ellas llamando a uno de los siguientes métodos definidos por la clase **CardLayout**:

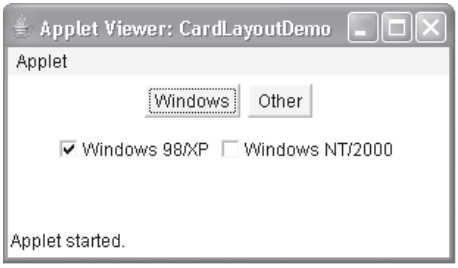
void first(Container *contenedor*)
void last(Container *contenedor*)
void next(Container *contenedor*)
void previous(Container *contenedor*)
void show(Container *contenedor*, String *nom*)

Aquí, *contenedor* es una referencia al contenedor (generalmente un panel) que contiene todas las cartas, y *nom* es el nombre de una carta o tarjeta. Al llamar al método **first( )** se muestra la primera carta o tarjeta, y para mostrar la última se llama al método **last( )**. Para mostrar la siguiente carta o tarjeta, se llama al método **next( )**. Para mostrar la carta o tarjeta anterior, se llama al método **previous( )**. Tanto **next( )** como **previous( )** rotan automáticamente las cartas. El método **show( )** visualiza la carta o tarjeta cuyo nombre se proporciona en *nom*.

El siguiente ejemplo crea dos tarjetas para poder seleccionar un sistema operativo. Con la primera tarjeta se visualizan los sistemas que operan bajo Windows, y con la segunda los que operan bajo Macintosh y Solaris.

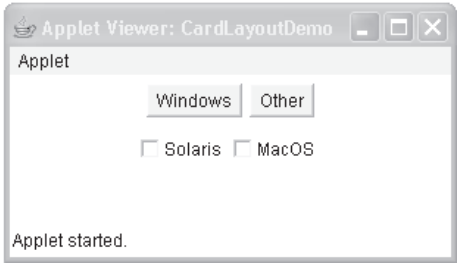
```
// Ejemplo de CardLayout.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="CardLayoutDemo" width=300 height=100>
  </applet>
*/

public class CardLayoutDemo extends Applet
  implements ActionListener, MouseListener {

  Checkbox WinXP , WinVista, solaris, mac;
  Panel osCards;
  CardLayout cardLO;
  Button Win, Otro;

  public void init() {
    Win = new Button("Windows");
    Otro = new Button("Otro");
    add(Win);
    add(Otro);

    cardLO = new CardLayout();
    osCards = new panel();
    osCards.setLayout(cardLO); // establece la organización del panel

    WinXP = new Checkbox("Windows XP", null, true);
    WinVista = new Checkbox("Windows Vista");
    solaris = new Checkbox("Solaris");
    mac = new Checkbox("MacOS");

    // añade los checkbox de Windows a un panel
    Panel winpan = new Panel();
    winPan.add(WinXP);
    winPan.add(WinVista);

    // Añade los checkbox de otros sistemas a un panel
    Panel otherpan = new Panel();
    otherPan.add(solaris);
    otherPan.add(mac);

    // añade los paneles al panel de las carta
    osCards.add(winPan, "Windows");
    osCards.add(otherPan, "Otro");

    // añade las cartas al panel principal del applet
    add(osCards);

    // registra eventos de acción
    Win.addActionListener(this);
    Otro.addActionListener(this);

    // registra los eventos de ratón
    addMouseListener(this);
  }

  // Para moverse a través de los paneles.
```

```
  public void mousePressed(MouseEvent me) {
    cardLO.next(osCards);
  }

  // Implementaciones vacías para los otros métodos de MouseListener.
  public void mouseClicked(MouseEvent me) {
  }
  public void mouseEntered(MouseEvent me) {
  }
  public void mouseExited(MouseEvent me) {
  }
  public void mouseReleased{MouseEvent me) {
  }

  public void actionPerformed(ActionEvent ae) {
    if(ae.getSource() == Win) {

      cardLO.show(osCards, "Windows");
    }
    else {
      cardLO.show(osCards, "Otro");
    }
  }
}
```



A continuación se muestra la salida generada por el applet **CardLayoutDemo**. Cada carta o tarjeta se activa pulsando su botón. También se puede pasar de carta en carta haciendo clic con el ratón.

GridBagLayout



Aunque los administradores de diseño anteriores son perfectamente aceptables para muchos usos, en algunas situaciones se requerirá que se tome un poco más de control sobre cómo los componentes son organizados. Una buena forma de hacer eso es utilizar un **GridBagLayout**. Lo que hace especial al **GridBagLayout** es que se puede especificar la posición relativa de los componentes, especificando su posición dentro de las celdas de la cuadrícula. La clave del **GridBagLayout** es que cada componente puede tener un tamaño diferente, y cada línea puede tener un número diferente de columnas. Este gestor de diseño toma su nombre del hecho de ser una colección de pequeñas cuadrículas juntas.

La ubicación y el tamaño de cada componente en el organizador están determinadas por un conjunto de restricciones ligadas a él. Las restricciones están contenidas en un objeto de tipo **GridBagConstraints**. Las restricciones incluyen la altura o la anchura de la celda y el lugar del componente, su alineación y su punto de anclaje dentro de la celda.

El procedimiento general para utilizar un organizador **GridBagLayout** es, en primer lugar, crear un nuevo objeto de tipo **GridBagLayout** y convertirlo en el organizador actual. Luego, se definen las restricciones que se desean aplicar a cada componente que será agregado en el organizador. Finalmente, se agregan los componentes al organizador.

Aunque **GridBagLayout** es un poco más complicado que los otros organizadores, es aún más sencillo de utilizar una vez que se comprende cómo funciona.

GridBagLayout define solo un constructor, el cual se muestra aquí:

GridBagLayout( )

GridBagLayout define varios métodos, de los cuales varios son protegidos y no son para uso general. Existe un método, sin embargo, que se debe utilizar: **setConstraints( )**, el cual se muestra a continuación:

void setConstraints(Component *comp*, GridBagConstraints *cons*)

Donde *comp* es el componente para el cual aplican las restricciones especificadas en *cons*. Este método define las restricciones que aplican a cada componente en el organizador.

La clave del éxito en el uso de **GridBagLayout** es la correcta definición de restricciones, las cuales son almacenadas en un objeto **GridBagConstrains**. **GridBagConstrains** define varios campos que se utilizan para definir los tamaños, acomodo, y espacio de un componente. Estos métodos se muestran en la Tabla 24-1. Algunos son descritos con más detalle más adelante.

Campo	Propósito
int anchor	Especifica la posición de un componente dentro de una celda. Por omisión es **GridBagConstraints.CENTER.**
int fill	Específica cómo un componente cambia de tamaño si el componente es más pequeño que su celda. Los valores válidos son **GridBagConstraints.NONE** (por omisión) **GridBagConstraints.HORIZONTAL, GridBagConstraints.VERTICAL, GridBagConstraints.BOTH**.
int gridheight	Especifica la altura de un componente en términos de celdas. Por omisión es 1.
int gridwidth	Especifica el ancho de un componente en términos de celdas. Por omisión es 1.
int gridx	Especifica la coordenada X de la celda a la cuál el componente será agregado. Por omisión el valor es **GridBagConstraints.RELATIVE**.
int gridy	Especifica la coordenada Y de la celda a la cuál el componente será agregado. Por omisión el valor es **GridBagConstraints.RELATIVE**.
Insets instes	Especifica la sangría. Por omisión todos los valores de insets son cero.
int ipadx	Específica un espacio horizontal extra que rodea a un componente dentro de una celda. El valor por omisión es 0.
int ipady	Específica un espacio vertical extra que rodea a un componente dentro de una celda. El valor por omisión es 0.
double weightx	Especifica un valor de peso que determina el espacio horizontal entre las celdas y los bordes de los contenedores que los mantienen. El valor por omisión es 0.0. Entre mayor sea el peso, más espacio será asignado. Si todos los valores son 0.0, el espacio extra es distribuido equitativamente entre los bordes de la ventana.
double weighty	Especifica un valor de peso que determina el espacio horizontal entre las celdas y los bordes de los contenedores que los mantienen. El valor por omisión es 0.0. Entre mayor sea el peso, más espacio será asignado. Si todos los valores son 0.0, el espacio extra es distribuido equitativamente entre los bordes de la ventana.

TABLA 24-1 Campos definidos por **GridBagConstraints**

GridBagConstraints también define varios campos estáticos que contienen valores estándares, tales como **GridBagConstraints**.**CENTER** y **GridBagConstraints**.**VERTICAL**.

Cuando un componente es más pequeño que su celda, se puede utilizar el campo **anchor** para especificar en dónde dentro de la celda se localizará el componente. Hay tres tipos de valores que se pueden dar a **anchor**. Los primeros son absolutos:

GridBagConstraints.CENTER	GridBagConstraints.SOUTH
GridBagConstraints.EAST	GridBagConstraints.SOUTHEAST
GridBagConstraints.NORTH	GridBagConstraints.SOUTHWEST
GridBagConstraints.NORTHEAST	GridBagConstraints.WEST
GridBagConstraints.NORTHWEST	

Como sus nombres lo implican, estos valores causan que los componentes sean colocados en el lugar especificado.

El segundo tipo de valores que pueden ser dados a **anchor** son valores relativos, lo cual significa que son relativos a la orientación del contenedor (esta definición no aplica si consideramos lenguajes no occidentales). Los valores relativos se muestran a continuación:

GridBagConstraints.FIRST_LINE_END	GridBagConstraints.LINE_END
GridBagConstraints.FIRST_LINE_START	GridBagConstraints.LINE_START
GridBagConstraints.LAST_LINE_END	GridBagConstraints.PAGE_END
GridBagConstraints.LAST_LINE_START	GridBagConstraints.PAGE_START

Sus nombres describen el lugar en que se sería colocado el componente.

El tercer tipo de valores que pueden ser dados a **anchor** fueron agregados por Java SE 6. Estos permiten posicionar componentes verticalmente de forma relativa a la línea base del renglón. Estos nuevos valores se muestran a continuación:

GridBagConstraints.BASELINE	GridBagConstraints.BASELINE_LEADING
GridBagConstraints.BASELINE_TRIALING	GridBagConstraints.ABOVE_BASELINE
GridBagConstraints.ABOVE_BASELINE_LEADING	GridBagConstraints.ABOVE_BASELINE_TRIALING
GridBagConstraints.BELOW_BASELINE	GridBagConstraints.BELOW_BASELINE_LEADING
GridBagConstraints.BELOW_BASELINE_TRAILING	

La posición horizontal puede estar también centrada, contra el borde superficial (LEADING) o contra el borde trasero (TRAILING).

Los campos **weightx** y **weighty** son muy importantes aunque confusos a primera vista. En general, sus valores determinan cuanto del espacio extra se asigna a cada fila y cada columna dentro del contenedor. Por omisión, estos valores son cero. Cuando todos los valores dentro de una fila o una columna son cero, el espacio extra es distribuido equitativamente entre los bordes de la ventana. Incrementando el peso, se incrementa el espacio asignado a la columna o renglón proporcionalmente con las otras columnas y filas. La mejor forma de entender cómo funcionan estos valores es experimentando con ellos un poco.

La variable **gridwidth** nos permite especificar el ancho de una celda en unidades celda. El valor por omisión es 1. Para especificar que un componente utiliza el espacio restante en una fila, se utiliza **GridBagConstraints.REMAINDER**. Para especificar que un componente utilice la última celda en la fila, se utiliza **GridBagConstraints.RELATIVE**. La variable **gridheight** funciona de la misma forma, sólo que en sentido vertical.

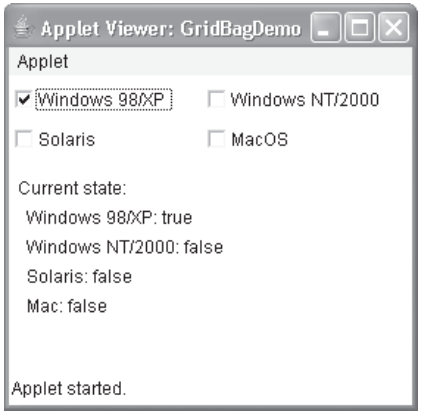
Se puede especificar un valor de relleno que sería utilizado para incrementar el tamaño mínimo de una celda. El relleno horizontal, se asigna al valor de **ipadx**. El relleno vertical, se asigna a **ipady**.

A continuación un ejemplo que utiliza **GridBagLayout** para demostrar varios de los puntos que se discutieron antes:


```
// Ejemplo de GridBagLayout.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="GridBagDemo" width=250 height=200>
  </applet>
*/

public class GridBagDemo extends Applet
  implements ItemListener {

  String msg = " ";
  Checkbox WinXP, WinVista, solaris, mac;

  public void init() {
    GridBagLayout gbag = new GridBagLayout();
    GridBagConstraints gbc = new GridBagConstraints();
    setLayout(gbag);

    // Crea los checkbox
    winXP = new Checkbox("Windows XP", null, true);
    winVista = new Checkbox("Windows Vista");
    solaris = new Checkbox("Solaris");
    mac = new Checkbox("MacOS");

    // Crea el GridBagLayout

    // Usa por omisión el peso 0 para la primera fila
    gbc.weightx = 1.0;  //usa el ancho de columna de 1
    gbc.ipadx = 200;  // relleno de 200 unidades
    gbc.insets = new Insets (4, 4, 0, 0);

    gbc.anchor = GridBagConstraints.NORTHEAST;

    gbc.gridwidth = GridBagConstraints.RELATIVE;
    gbag.setConstraints(winXP, gbc);

    gbc.gridwidth = GridBagConstraints.REMAINDER;
    gbag.setConstraints(winVista, gbc);

    // Para la segunda fila se da un peso de 1.
    gbc.weighty = 1.0;
```

```
    gbc.gridwidth = GridBagConstraints.RELATIVE;
    gbag.setConstraints (solaris, gbc);

    gbc.gridwidth = GridBagConstraints.REMAINDER;
    gbag.setConstraints (mac, gbc);

    //Agrega los componentes
    add(winXP);
    add(winVista);
    add(solaris);
    add(mac);

    //Registra los eventos de cada elemento registrado
    winXP.addItemListener(this);
    winVista.addItemListener(this);
    solaris.addItemListener(this);
    mac.addItemListener(this);
  }

  //Repinta cuando el estado de un checkbox cambia
  public void itemStateChanged(ItemEvent ie){
  repaint();
  }

  // Muestra el estado actual de los checkbox
  public void paint (Graphics g) {
    msg = "Estado actual: ";
    g.drawString(msg, 6, 80);
    msg = " Windows XP: " + winXP.getState( );
    g.drawString(msg, 6, 100);
    msg = " Windows Vista: " + winVista.getState( );
    g.drawString(msg, 6, 120);
    msg = " Solaris: " + solaris.getState( );
    g.drawString(msg, 6, 140);
    msg = " Mac:" +mac.getState( );
    g.drawString(msg, 6, 160);
  }
}
```

La salida del programa se muestra a continuación:



En este gestor de organización, los checkbox que representan a cada sistema operativo están organizados en una cuadrícula de 2x2. Cada celda tiene un relleno horizontal de 200. Cada componente tiene un margen pequeño de 4 unidades arriba y a la izquierda. El ancho de la columna está fijado en 1, lo cual causa que todo el espacio extra horizontal sea distribuido equitativamente entre las columnas. La primera fila utiliza el peso por omisión que es 0; la segunda tiene un peso de 1. Eso significa que cualquier espacio extra vertical se agrega a la segunda fila.

GridBagLayout es un organizador poderoso. Vale la pena tomar un poco de tiempo para experimentarlo y probarlo. Una vez que se comprende lo que hacen las propiedades, se puede utilizar el **GridBagLayout** para organizar los componentes con mayor grado de precisión.

Barras de menú y menús

Una ventana de nivel superior puede tener asociada una barra de menú. Una barra de menú muestra una lista de menús desplegables. Este concepto se implementa en AWT con las clases: **MenuBar**, **Menu**, y **MenuItem**. En general, una barra de menú tiene uno o más objetos **Menu**. Cada objeto **Menu** tiene una lista de objetos **MenuItem**. Cada objeto **MenuItem** representa algo que el usuario puede seleccionar. Debido a que **Menu** es una subclase de **MenuItem**, se puede crear una jerarquía de submenús anidados. También se pueden incluir elementos de menú que se puedan seleccionar. Estas opciones de menú son del tipo **CheckboxMenuItem** y muestran una marca al lado de ellos cuando se seleccionan.

Para crear una barra de menú, primero hay que crear una instancia de **MenuBar**. Esta clase sólo define el constructor por omisión. Después, se crean instancias de **Menu** que definan las selecciones que se muestran en la barra. Los constructores de **Menu** son los siguientes:

Menu( ) throws HeadlessException
Menu(String *nom*) throws HeadlessException
Menu(String *nom*, boolean *mov*) throws HeadlessException

Donde, *nom* especifica el nombre del menú de selección. Si *mov* es **true**, el menú puede ser movido y flotar libremente. En caso contrario, permanecerá siempre junto a la barra de menú (los menús removibles son dependientes de la implementación). El primer constructor crea un menú vacío.

Los elementos individuales del menú son de la clase **MenuItem**. Esta clase define los siguientes constructores:

MenuItem( ) throws HeadlessException
MenuItem(String *nom*) throws HeadlessException
MenuItem(String *nom*, MenuShortcut *k*) throws HeadlessException

Aquí, *nom* es el nombre que se muestra en el menú, y *k* es la tecla rápida para ese elemento.

Se puede habilitar o deshabilitar un elemento del menú utilizando el método **setEnabled( )**, que tiene el siguiente formato:

void setEnabled(boolean *e*)

Si el argumento *e* es **verdadero**, el elemento del menú será habilitado. Si es **falso**, será deshabilitado.

Se puede determinar el estado de un elemento llamando a **isEnabled( )**, que tiene el siguiente formato:

boolean isEnabled( )

isEnabled( ) devuelve **true** si el elemento del menú está habilitado. En caso contrario, devuelve **false**.

Para cambiar el nombre de un elemento del menú se llama al método **setLabel( )**. Para obtener el nombre se llama al método **getLabel( )**. Estos métodos tienen la siguiente forma:

void setLabel(String *nom*)
String getLabel( )

Aquí, *nom* es el nuevo nombre del elemento del menú. El método **getLabel( )** devuelve el nombre actual.

Se puede crear un elemento de menú seleccionable utilizando una subclase de **MenuItem** llamada **CheckboxMenuItem**, que tiene los siguientes constructores:

CheckboxMenuItem( ) throws HeadlessException
CheckboxMenuItem(String *nom*) throws HeadlessException
CheckboxMenuItem(String *nom*, boolean *e*) throws HeadlessException

Aquí, *nom* es el nombre que se muestra en el menú. Los elementos que se pueden seleccionar funcionan como conmutadores. Cada vez que se selecciona, cambia de estado. En los dos primeros formatos, la entrada que se puede seleccionar no está seleccionada, pero en el tercer formato, si *e* es verdadero, esa entrada se selecciona. En caso contrario, no.

Se puede obtener el estado de un elemento seleccionable llamando a **getState( )**. Se le puede asignar un estado utilizando el método **setState( )**. Estos métodos tienen los siguientes formatos:

boolean getState( )
void setState(boolean *e*)

Si el elemento está seleccionado, el método **getState( )** devuelve **true**. En caso contrario, devuelve **false**. Para seleccionar un elemento, se pasa **true** al método **setState( )** y para quitar la selección se pasa **false**.

Una vez que se ha creado un elemento de menú, hay que añadirlo a un objeto **Menu** utilizando el método **add( )**, que tiene el siguiente formato:

MenuItem add(MenuItem *item*)

Donde *item* es el elemento que se va a añadir. Los elementos se añaden al menú en el orden en que se llama al método **add( )**. Este método devuelve el elemento añadido.

Una vez que se han añadido todos los elementos a un objeto **Menu**, se puede añadir ese objeto a la barra de menú utilizando la siguiente versión del método **add( )** definida por **MenuBar**:

Menu add(Menu *menu*)

Aquí, *menu* es el menú que se añade. El método devuelve el menú.

Los menús sólo generan eventos cuando se selecciona un elemento de la clase **MenuItem** o **CheckboxMenuItem**. No se genera ningún evento cuando, por ejemplo, se accede a la barra de menú para mostrar un menú desplegable. Cada vez que se selecciona un elemento de menú, se genera un objeto de tipo **ActionEvent**. Por omisión, la cadena de comando de acción es el nombre del elemento del menú. Sin embargo, se puede especificar un comando de acción diferente llamando al método **setActionCommand( )** en el elemento del menú. Cada vez que se selecciona o se deselecciona un elemento de menú de la clase **Checkbox**, se genera un objeto **ItemEvent**. Por tanto, hay que implementar las interfaces **ActionListener** e **ItemListener** para gestionar esos eventos de menú.

El método **getItem( )** de **ItemEvent** devuelve una referencia al elemento que ha generado ese evento. El formato para este método es el siguiente:

Object getItem( )

A continuación se muestra un ejemplo que añade una serie de menús anidados a una ventana. En la ventana se muestra el elemento seleccionado. También se muestra el estado de los dos elementos de menú de la clase Checkbox.

```
// Ejemplo de menús.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="MenuDemo" width=250 height=250>
  </applet>
*/

// Se crea una subclase de Frame
class MenuFrame extends Frame {
  String msg = "";
  CheckboxMenuItem debug, test;

  MenuFrame(String title) {
    super(title);

    // se crea una barra de menú y se añade a la ventana
    MenuBar mbar = new MenuBar();
    setMenuBar(mbar);

    // se crean los elementos de menú
    Menu file = new Menu("Archivo");
    MenuItem iteml, item2, item3, item4, item5;
    file.add(iteml = new MenuItem("Nuevo..."));
    file.add(item2 = new MenuItem("Abrir..."));
    file.add(item3 = new MenuItem("Cerrar"));
    file.add(item4 = new MenuItem("-"));
    file.add(item5 = new MenuItem("Salir..."));
    mbar.add(file);

    Menu edit = new Menu("Edición");
    MenuItem item6, item7, item8, item9;
    edit.add(item6 = new MenuItem("Cortar")) ;
    edit.add(item7 = new MenuItem("Copiar"));
    edit.add(item8 = new MenuItem("Pegar"));
    edit.add(item9 = new MenuItem("-"));

    Menu sub = new Menu("Especial");
    MenuItem iteml0, itemll. item12;
    sub.add(item10 = new MenuItem("Primero"));
    sub.add(itemll = new MenuItem("Segundo"));
    sub.add(item12 = new MenuItem("Tercero"));
    edit.add(sub);

    // estos son los elementos de menú que se pueden seleccionar
    debug = new CheckboxMenuItem("Depurar");
    edit.add(debug);
    test = new CheckboxMenuItem("Prueba");
    edit.add(test);

    mbar.add(edit);

    // se crea un objeto para gestionar eventos
    MyMenuHandler handler = new MyMenuHandler(this);
    // se registra para recibir esos eventos
    item1.addActionListener(handler);
    item2.addActionListener(handler);
```

```
    item3.addActionListener(handler);
    item4.addActionListener(handler) ;
    item5.addActionListener(hand1er);
    item6.addActionListener(handler);
    item7.addActionListener(handler);
    item8.addActionListener(handler);
    item9.addActionListener(handler);
    item10.addActionListener(handler);
    item11.addActionListener(handler) ;
    item12.addActionListener(handler);
    debug.addItemListener(handler);
    test.addItemListener(handler);

    // se crea un objeto para gestionar eventos de la ventana
    MyWindowAdapter adapter = new MyWindowAdapter(this);
    // se registra para recibir esos eventos
    addWindowListener(adapter);
  }

  public void paint(Graphics g) {
    g.drawString(msg, 10, 200);

  if(debug.getState())
    g.drawString("Depuración activada.", 10, 220);
  else
    g.drawString("Depuración desactivada.", 10, 220);

  if(test.getState())
    g.drawString("Prueba activada.", 10, 240);
  else
    g.drawString("Prueba desactivada.", 10, 240);
  }
}

class MyWindowAdapter extends WindowAdapter {
  MenuFrame menuFrame;
  public MyWindowAdapter(MenuFrame menuFrame) {
    this.menuFrame = menuFrame;
  }
  public void windowClosing(WindowEvent we) {
    menuFrame.setVisible(false);
  }
}

class MyMenuHandler implements ActionListener, ItemListener {
  MenuFrame menuFrame;
  public MyMenuHandler(MenuFrame menuFrame) {
    this.menuFrame = menuFrame;
  }
  // Gestión de eventos de acción
  public void actionPerformed(ActionEvent ae) {
      String msg = "Usted ha seleccionado";
      String arg = (String)ae.getActionCornmand();
      if(arg.equals("Nuevo..."))
        msg += "Nuevo.";
      else if(arg.equals("Abrir …"))
```

```
      msg += "Abrir.";
    else if (arg.equals ("Cerrar"))
      msg += "Cerrar.";
    else if(arg.equals("Salir…"))
      msg += "Salir.";
    else if(arg.equals("Edición"))
      msg += "Edición.";
    else if(arg.equals("Cortar"))
      msg += "Cortar.";
    else if(arg.equals("Copiar"))
      msg += "Copiar.";
    else if(arg.equals("Pegar"))
      msg += "Pegar.";
    else if(arg.equals("Primero"))
      msg += "Primero.”;
    else if(arg.equals("Segundo"))
      msg += "Segundo.";
    else if(arg.equals("Tercero"))
      msg += "Tercero.";
    else if(arg.equals("Depurar "))
      msg += "Depurar.";
    else if(arg.equals("Prueba"))
      msg += "Prueba.";
    menuFrame.msg = msg;
    menuFrame.repaint();
  }
  // Gestión de eventos de los elementos
  public void itemStateChanged(ItemEvent ie) {
    menuFrame.repaint();
  }
}

// Se crea la ventana.
public class MenuDemo extends Applet {
  Frame f;

  public void init() {
    f = new MenuFrame("Menu Demo");
    int width = Integer.parseInt(getParameter("width"));
    int height = Integer.parseInt(getParameter("height"));

    setSize(new Dimension(width, height));

    f.setSize(width, height);
    f.setVisible(true);
  }

  public void start() {
    f.setVisible(true);
  }

  public void stop() {
    f.setVisible(false);
  }
}
```

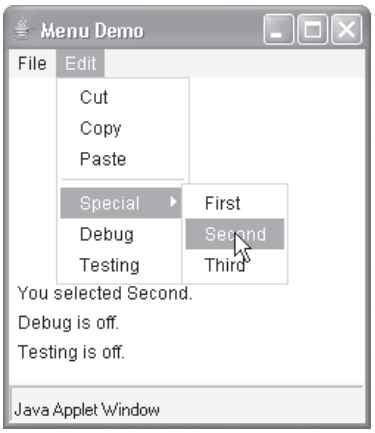


FIGURA 24-8 Ejemplo de salida del applet **MenuDemo**

La salida del applet **MenuDemo** se muestra en la Figura 24-8.

Hay otra clase relacionada con los menús que es interesante: la clase **PopupMenu**. Funciona de manera similar a **Menu** pero genera un menú que se puede mostrar en una posición determinada. **PopupMenu** es una alternativa útil y flexible para ciertas situaciones donde se trabaja con menús.

Cuadros de diálogo

A menudo se suelen utilizar *cuadros de diálogo* para agrupar un conjunto de controles relacionados. Los cuadros de diálogo se utilizan principalmente para obtener entradas del usuario. Son similares a las ventanas, excepto en que los cuadros de diálogo son siempre ventanas hijas de una ventana de nivel superior. Los cuadros de diálogo no tienen barras de menú, pero en general funcionan igual que las ventanas; por ejemplo, se les puede añadir controles de la misma manera que se añaden controles a una ventana. Los cuadros de diálogo pueden ser de tipo modal o no modal. Cuando está activo un cuadro de diálogo *modal*, todas las entradas se dirigen a él hasta que se cierre. Esto significa, que no se pueden acceder otras partes del programa hasta que se cierre el cuadro de diálogo. Cuando se activa un cuadro de diálogo *no modal*, la entrada se puede dirigir a otras ventanas del programa, esto es, las otras partes del programa permanecen activas y accesibles. Los cuadros de diálogo son de la clase **Dialog**. Los constructores que más se utilizan son los siguientes:

Dialog(Frame *parentWindow*, boolean *modo*)
Dialog(Frame *parentWindow*, String *titulo*, boolean *modo*)

Donde *parentWindow* es el propietario del cuadro de diálogo. Si el parámetro *modo* es **true**, el cuadro de diálogo es *modal*. En caso contrario, es *no modal*. El título del cuadro de diálogo se pasa en el parámetro *titulo*. Generalmente, se crean subclases de la clase **Dialog** para añadir la funcionalidad requerida por nuestra aplicación.

A continuación se muestra una versión modificada del programa anterior que visualiza un cuadro de diálogo no modal cuando se elige la opción Nuevo del menú. Obsérvese que cuando se cierra el cuadro de diálogo, se llama al método **dispose( )**. Este método está definido en la

clase **Window**, y libera todos los recursos del sistema asociados con la ventana del cuadro de diálogo.

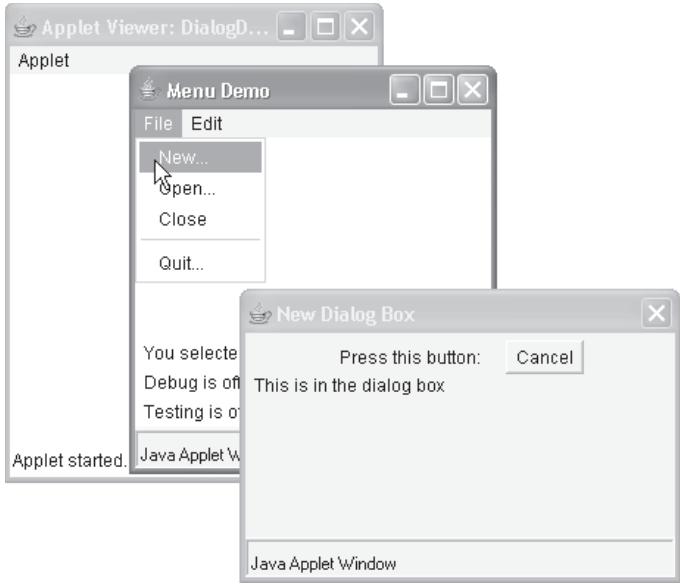
```
// Ejemplo de cuadros de diálogo.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="DialogDemo" width=250 height=250>
</applet>
*/

// Se crea una subclase de Dialog.
class SampleDialog extends Dialog implements ActionListener {
  SampleDialog(Frame parent, String title) {
    super (parent, title, false);
    setLayout(new FlowLayout());
    setSize(300, 200);

    add(new Label("Presione este botón:"));
    Button b;
    add{b = new Button("Cancelar"));
    b.addActionListener(this);
  }

  public void actionPerformed(ActionEvent ae) {
    dispose();
  }

  public void paint(Graphics g) {
    g.drawString("Esto está en el cuadro de diálogo", 10, 70);
  }
}

// Se crea una subclase de Frame.
class MenuFrame extends Frame {
  String msg = "";
  CheckboxMenuItem debug, test;

  MenuFrame(String title) {
    super (title) ;

    // se crea una barra de menú y se añade a la ventana
    MenuBar mbar = new MenuBar();
    setMenuBar(mbar);

    // se crean los elementos de menú
    Menu file = new Menu("Archivo");
    MenuItem item1, item2, item3, item4;
    file.add(item1 = new MenuItem("Nuevo..."));
    file.add(item2 = new MenuItem("Abrir..."));
    file.add(item3 = new MenuItem("Cerrar"));
    file.add(new MenuItem("-"));
    file.add(item4 = new MenuItem("Salir…"));
    mbar.add(file);

    Menu edit = new Menu("Edición");
    MenuItem item5, item6, item7;
    edit.add(item5 = new MenuItem("Cortar"));
```

```
    edit.add(item6 = new MenuItem("Copiar"));
    edit.add(item7 = new MenuItem("Pegar"));
    edit.add(new MenuItem("-"));

    Menu sub = new Menu("Especial", true);
    MenuItem item8, item9, item10;
    sub.add(item8 = new MenuItem("Primero"));
    sub.add(item9 = new MenuItem("Segundo"));
    sub.add(item10 = new MenuItem("Tercero"));
    edit.add(sub);

    // estos son los elementos de menú que se pueden seleccionar
    debug = new CheckboxMenuItem("Depurar");
    edit.add(debug);
    test = new CheckboxMenuItem("Prueba");
    edit.add(test);

    mbar.add(edit) ;

    // se crea un objeto para gestionar eventos
    MyMenuHandler handler = new MyMenuHandler(this);
    // se registra para recibir esos eventos
    item1.addActionListener(handler);
    item2.addActionListener(handler);
    item3.addActionListener(handler);
    item4.addActionListener(handler);
    item5.addActionListener(handler);
    item6.addActionListener(handler);
    item7.addActionListener(handler);
    item8.addActionListener(hand1er);
    item9.addActionListener(handler);
    item10.addActionListener(hand1er);
    debug.addItemListener(handler);
    test.addItemListener(handler);

    // se crea un objeto para gestionar eventos de ventana
    MyWindowAdapter adapter = new MyWindowAdapter(this);
      // se registra para recibir esos eventos
    addWindowListener(adapter);
  }
  public void paint(Graphics g) {
    g.drawString(msg, 10, 200);

    if(debug.getState())
      g.drawString("Depuración activada.", 10, 220);
    else
      g.drawString("Depuración desactivada.", 10, 220);

    if(test.getState())
      g.drawString ("Prueba activada.", 10, 240);
    else
      g.drawString("Prueba desactivada.", 10, 240);
    }
  }
```

```
class MyWindowAdapter extends WindowAdapter {
  MenuFrame menuFrame;
  pub1ic MyWindowAdapter(MenuFrame menuFrame) {
    this.menuFrame = menuFrame;
  }
  public void windowClosing(WindowEvent we) {
    menuFrame.dispose();
  }
}

class MyMenuHand1er implements ActionListener, ItemListener {
  MenuFrame menuFrame;
  public MyMenuHand1er(MenuFrame menuFrame) {
    this.menuFrame = menuFrame;
  }
  // Gestiona los eventos de acción
  public void actionPerformed(ActionEvent ae) {
    String msg = "Usted ha seleccionado ";
    String arg = (String)ae.getActionCornmand();
    // Activa un cuadro de diálogo cuando se selecciona Nuevo.
    if (arg.equals ("Nuevo...")) {
      msg += "Nuevo.";
      SampleDialog d = new SampleDialog, (menuFrame "Nueva caja de diálogo");
      d.setVisible(true);
    }
    // Intente definir otros cuadros de diálogo para estas opciones.
    else if (arg .equals ("Abrir..."))
      rnsg+= "Abrir.";
    else if(arg.equals("Cerrar"))
      msg += "Cerrar.";
    el se if(arg.equals("Salir..."))
      msg += "Salir.";
    else if(arg.equals("Edición"))
      msg += "Edición.";
    else if(arg.equals("Cortar"))
      msg+= "Cortar";
    else if(arg.equals("Copiar"))
      msg += "Copiar";
    else if (arg.equals("Pegar") )
      msg += "Pegar.";
    else if(arg.equals("Primero"))
      msg += "Primero.";
    else if (arg.equals("Segundo") )
      msg += "Segundo.";
    else if(arg.equals("Tercero"))
      msg += "Tercero.";
    else if(arg.equals("Depurar"))
      msg += "Depurar.";
    else if(arg.equals("Prueba"))
      msg += "Prueba.";
    menuFrame.msg = msg;
    menuFrame.repaint() ;
  }
```

```
  public void itemStateChanged(ItemEvent ie) {
       menuFrame.repaint();
  }
}

// Crea la ventana del marco.
public class DialogDemo extends Applet {
  Frame f;

  public void init() {
    f = new MenuFrame("Menu Demo");
    int width = Integer.parseInt(getParameter("width"));
    int height = Integer.parseInt(getParameter("height"));

    setSize(width, height);

    f.setSize(width, height);
    f.setVisible(true);
  }

  public void start() {
    f.setVisible(true);
  }

  public void stop() {
    f.setVisible(false);
  }
}
```

La salida del applet **DialogDemo** es la siguiente:



NOTA *El lector puede intentar, por cuenta propia, definir cuadros de diálogo para las otras opciones presentadas en los menús.*

FileDialog

Java proporciona un cuadro de diálogo preconstruido que permite al usuario seleccionar un archivo. Para crear un cuadro de diálogo de archivo, sólo hay que crear un objeto de la clase **FileDialog**. Con ello se visualiza un cuadro de diálogo de archivo. Normalmente, éste es el cuadro de diálogo que presenta el sistema operativo. La clase **FileDialog** tiene estos constructores:

FileDialog(Frame *parent*)
FileDialog(Frame *parent*, String *nom*)
FileDialog(Frame *parent*, String *nom*, int *p*)

Donde, *parent* es el propietario de la caja de diálogo, y *nom* es el nombre que aparece en la barra de título del cuadro. Si se omite *nom*, no aparecerá ningún título. Si *p* tiene el valor **FileDialog.LOAD**, el cuadro selecciona un archivo para lectura. Si *p* es **FileDialog.SAVE**, el cuadro selecciona un archivo para escritura. Si *p* se omite, el cuadro selecciona a un archivo para lectura.

La clase **FileDialog** tiene métodos que permiten determinar el nombre y la ruta del archivo seleccionado por el usuario. A continuación se presentan dos ejemplos:

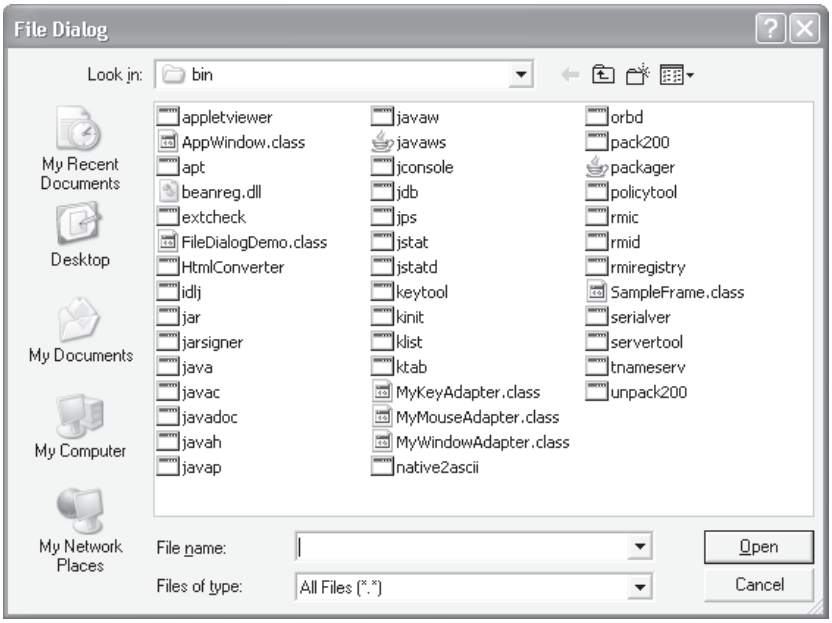
String getDirectory( )
String getFile( )

Estos métodos devuelven el directorio y el nombre del archivo, respectivamente. El siguiente programa ejemplifica el uso de cuadro de diálogo para selección de archivos:

```
/* Ejemplo de FileDialog.

  Esto no es un applet, sino una aplicación.
*/
import java.awt.*;
import java.awt.event.*;

// Crea una subclase de Frame
class SampleFrame extends Frame {
  SampleFrame(String title) {
    super(title);

    //remueve la ventana cuando se cierra
    addWindowListener(new WindowAdapter() {
      public void windowClosing (WindowEvent we) {
        System.exit(0);
      }
    });

  }
}

// Crea la ventana
class FileDialogDemo {
  public static void main(String args[]) {
    //Crea una ventana que poseerá el cuadro de diálogo
    Frame f = new SampleFrame ("Ejemplo de cuadro de diálogo para selección de
    archivos ");
    f.setVisible(true);
    f.setSize(100, 100);
FileDialog fd = new FileDialog(f, "Selección de archivos");
```

```
    fd.setVisible(true);
  }
}
```

La salida generada por este programa es la siguiente (la apariencia exacta del cuadro de diálogo puede variar):



Gestión de eventos extendiendo los componentes AWT

No se puede dar por concluida nuestra revisión al AWT sin estudiar la gestión de eventos extendiendo los componentes AWT. En el Capítulo 22 se estudió el modelo de delegación de eventos, y todos los programas de este libro han utilizado ese diseño. Pero Java también permite gestionar eventos creando subclases de los componentes AWT. Hacer eso permite gestionar los eventos de manera parecida a cómo se gestionaban con la versión original 1.0 de Java. Obviamente, no se recomienda esta técnica, ya que tiene las mismas desventajas que el modelo de eventos del Java 1.0; la más importante de ellas es la ineficacia. Se describe en este apartado la gestión de eventos extendiendo componentes AWT para dar a conocer su existencia. Sin embargo, esta técnica no se va a utilizar en ninguna otra sección de este libro.

Para ampliar un componente AWT, hay que llamar al método **enableEvents( )** de **Component**. Su formato general es el siguiente:

protected final void enableEvents(long *eventMask*)

El argumento *eventMask* es un bit máscara que define los eventos que se van a incluir en ese componente. La clase **AWTEvent** define constantes **enteras** para construir esa máscara, algunas de las cuales se muestran a continuación:

ACTION_EVENT _MASK	KEY_EVENT_MASK
ADJUSTMENT_EVENT_MASK	MOUSE_EVENT_MASK
COMPONENT _EVENT_MASK	MOUSE_MOTION_EVENT _MASK
CONTAINER_EVENT_MASK	MOUSE_WHEEL_EVENT_MASK
FOCUS_EVENT_MASK	TEXT_EVENT_MASK
INPUT_METHOD_EVENT_MASK	WINDOW_EVENT_MASK
ITEM_EVENT_MASK	

También hay que sobrescribir el método apropiado de una de las superclases para procesar el evento. En la Tabla 24-2 se muestran los métodos más comunes que se utilizan y las clases que los proporcionan.

Las secciones siguientes proporcionan algunos programas sencillos que muestran cómo extender algunos componentes AWT.

Extender Button

El siguiente programa crea un applet que visualiza un botón que tiene la etiqueta"Botón de Prueba". Cuando se pulsa el botón, aparece en la línea de estado del visor de applets o del navegador la cadena"evento de acción:", seguido de un contador con el número de veces que se ha presionado el botón.

El programa tiene una clase de nivel superior que se llama **ButtonDemo2** que extiende **Applet**. Se define una variable estática entera que se llama *i* y se inicializa a cero. Esta variable va a guardar el número de veces que se pulsa el botón. El método **init( )** crea una instancia de **MyButton** y lo añade al applet.

Clase	Métodos de Proceso
Button	processActionEvent( )
Checkbox	processItemEvent( )
CheckboxMenuItem	processItemEvent( )
Choice	processItemEvent( )
Component	processComponentEvent( ), processFocusEvent( ), processKeyEvent( ), processMouseEvent( ), processMouseMotionEvent( ), processMouseWheelEvent( )
List	processActionEvent( ), processItemEvent( )
MenuItem	processActionEvent( )
Scrollbar	processAdjustmentEvent( )
TextComponent	processTextEvent( )

TABLA 24-2 Métodos de proceso del evento

MyButton es una clase interna que extiende **Button**. Su constructor utiliza **super** para pasar la etiqueta del botón al constructor de la superclase. Llama al método **enableEvents( )**, por lo que ese objeto puede recibir los eventos de acción. Cuando se genera un evento de acción, se llama al método **processActionEvent( )**. Ese método visualiza una cadena en la línea de estado y llama al método **processActionEvent( )** de la superclase. Debido a que **MyButton** es una clase interna, tiene acceso directo al método **showStatus( )** de la clase **ButtonDemo2.**

```
/*
* <applet code=ButtonDemo2 width=200 height=100>
* </applet>
*/
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class ButtonDemo2 extends Applet {
  MyButton myButton;
  static int i = 0;
  public void init() {
    myButton = new MyButton("Botón de Prueba");
    add(myButton);
  }
  class MyButton extends Button {
    public MyButton(String etiqueta)
      super(etiqueta);
      enableEvents(AWTEvent.ACTION_EVENT_MASK);
    }
    protected void processActionEvent(ActionEvent ae) {
      showStatus ("evento de acción: " + i++) ;
      super.processActionEvent(ae);
    }
  }
}
```

Extender Checkbox

El siguiente programa crea un applet que visualiza tres checkbox con etiquetas "Elemento 1", "Elemento 2" y "Elemento 3". Cuando se selecciona o se deselecciona un checkbox, se visualiza una cadena que contiene el nombre y el estado del checkbox en la línea de estado del visor de applets o del navegador.

El programa tiene una clase de nivel superior llamada **CheckboxDemo2** que extiende **Applet**. Su método **init( )** crea tres instancias de **MyCheckbox** y las añade al applet. **MyCheckbox** es una clase interna que extiende **Checkbox**. Su constructor utiliza **super** para pasar la etiqueta del checkbox al constructor de la superclase. Llama a **enableEvents( )**, por lo que este objeto puede recibir eventos de los elementos añadidos. Cuando se genera un evento de elemento, se llama al método **processItemEvent( )**. Ese método visualiza una cadena en la línea de estado y a continuación llama al método **processItemEvent( )** de la superclase.

```
/*
* <applet code=CheckboxDemo2 width=300 height=100>
* </applet>
*/
import java.awt.*;
```

```
import java.awt.event.*;
import java.applet.*;

public class CheckboxDemo2 extends Applet {
  MyCheckbox myCheckbox1, myCheckbox2, myCheckbox3;
  public void init() {
    myCheckbox1 = new MyCheckbox("Elemento 1");
    add(myCheckbox1);
    myCheckbox2 = new MyCheckbox("Elemento 2");
    add(myCheckbox2);
    myCheckbox3 = new MyCheckbox("Elemento 3");
    add(myCheckbox3);
  }
  class MyCheckbox extends Checkbox {
    public MyCheckbox(String etiqueta) {
      super(etiqueta); .
      enableEvents(AWTEvent.ITEM_EVENT_MASK);
    }
    protected void processItemEvent(ItemEvent ie) {
      showStatus("Nombre/Estado del checkbox: " + getLabel() +
                 "/" + getState());
      super.processItemEvent(ie);
    }
  }
}
```

Extender CheckboxGroup

El siguiente programa rehace el ejemplo anterior de checkbox para que los checkbox formen un grupo de checkbox. De esta forma, sólo se puede seleccionar una de los checkbox al mismo tiempo.

```
 /*
* <applet code=CheckboxGroupDemo2 width=300 height=100>
* </applet>
*/
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class CheckboxGroupDemo2 extends Applet {
  CheckboxGroup cbg;
  MyCheckbox myCheckbox1, myCheckbox2, myCheckbox3;
  public void init() {
    cbg = new CheckboxGroup();
    myCheckbox1 = new MyCheckbox("Elemento 1");
    add(myCheckbox1);
    myCheckbox2 = new MyCheckbox("Elemento 2");
    add(myCheckbox2);
    myCheckbox3 = new MyCheckbox("Elemento 3");
    add(myCheckbox3);
  }
  class MyCheckbox extends Checkbox {
    public MyCheckbox(String etiqueta, CheckboxGroup cbg,
                      boolean flag) {
      super(etiqueta, cbg, flag);
```

```
    enableEvents(AWTEvent.ITEM_EVENT_MASK);
  }
  protected void processItemEvent(ItemEvent ie) {
    showStatus("Nombre/Estado del checkbox: " + getLabel() +
               "/" + getState());
    super.processItemEvent(ie);
  }
 }
}
```

Extender Choice

El siguiente programa crea un applet que visualiza una lista de opciones con elementos que tienen por etiqueta "Rojo", "Verde"y "Azul". Cuando se selecciona una entrada, se visualiza una cadena que contiene el nombre del color en la línea de estado del visor de applets o del navegador.

La clase de nivel superior llamada **ChoiceDemo2** que extiende **Applet**. Su método **init( )** crea un elemento de tipo **MyChoice** y lo añade al applet. **MyChoice** es una clase interna que extiende **Choice**. En **MyChoice** se llama al método **enableEvents( )**, por lo que el objeto puede recibir eventos de elemento. Cuando se genera un evento de elemento, se llama al método **processItemEvent( )**. Ese método visualiza una cadena en la línea de estado y llama al método **processItemEvent( )** de la superclase.

```
/*
* <applet code=ChoiceDemo2 width=300 height=100>
* </applet>
*/
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class ChoiceDemo2 extends Applet {
  MyChoice choice;
  public void init() {
    choice = new MyChoice();
    choice.add("Rojo");
    choice.add("Verde");
    choice.add("Azul");
    add(choice);
  }
  class MyChoice extends Choice {
    public MyChoice() {
      enableEvents(AWTEvent.ITEM_EVENT_MASK);
    }
    protected void processItemEvent(ItemEvent ie) {
      showStatus("Opción elegida: " + getSelectedItem());
      super.processItemEvent(ie);
    }
  }
}
```

Extender List

El siguiente programa modifica el ejemplo anterior para utilizar una lista en lugar de un menú de opciones. La clase de nivel superior llamada **ListDemo2** extiende **Applet**. Su método **init( )** crea una lista y la añade al applet. **MyList** es una clase interna que extiende **List**.

En esta clase se llama al método **enableEvents( )**, por lo que este objeto puede recibir eventos tanto de elemento como de acción. Cuando se selecciona o se deselecciona una entrada, se llama al método **processItemEvent( )**. Cuando se hace doble clic en una entrada, también se llama al método **processActionEvent( )**. Ambos métodos visualizan una cadena y pasan el control a la superclase.

```
/*
* <applet code=ListDemo2 width=300 height=100>
* </applet>
*/
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class ListDemo2 extends Applet {
  MyList list;
  public void init() {
    list = new MyList();
    list.add("Rojo");
    list.add("Verde");
    list.add("Azul");
    add(list);
  }
  class MyList extends List {
    public MyList() {
      enableEvents(AWTEvent.ITEM_EVENT_MASK |
                   AWTEvent.ACTION_EVENT_MASK);
    }
    protected void processActionEvent(ActionEvent ae) {
      showStatus("Evento de acción: " + ae.getActionCommand());
      super.processActionEvent(ae);
    }
    protected void processItemEvent(ItemEvent ie) {
      showStatus("Evento de elemento: " + getSelectedItem());
      super.processItemEvent(ie);
    }
  }
}
```

Extender Scrollbar

El siguiente programa crea un applet que visualiza una barra de desplazamiento. Cuando se manipula este control, se visualiza una cadena en la línea de estado del visor de applets o del navegador. En la cadena se incluye el valor representado por la barra de desplazamiento.

La clase de nivel superior llamada **ScrollbarDemo2** extiende **Applet**. Su método **init( )** crea una barra de desplazamiento y la añade al applet. La clase **MyScrollbar** es una clase interna que extiende **Scrollbar**. Esta clase llama al método **enableEvents( )** por lo que este objeto puede recibir eventos de ajuste. Cuando se manipula una barra de desplazamiento, se llama al método **processAdjustmentEvent( )**. Cuando se selecciona una entrada, se llama al método **processAdjustmentEvent( )**. Ese método visualiza una cadena y pasa el control a la superclase.

```
/*
* <applet code=ScrollbarDemo2 width=300 height=100>
* </applet>
```

```
*/
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class ScrollbarDemo2 extends Applet {
  MyScrollbar myScrollbar;
  public void init() {
    myScrollbar = new MyScrollbar(Scrollbar.HORIZONTAL,
                                  0, 1, 0, 100);

    add(myScrollbar);
  }
  class MyScrollbar extends Scrollbar {
    public MyScrollbar(int style, int initial, int thumb,
                       int min, int max) {
      super (style, initial, thumb, min, max);
      enableEvents(AWTEvent.ADJUSTMENT_EVENT_MASK);
    }
    protected void processAdjustmentEvent(AdjustmentEvent ae) {
      showStatus("Evento de ajuste: " + ae.getValue());
      setValue(getValue());
      super.processAdjustmentEvent(ae);
    }
  }
}
```

25 CAPÍTULO

Imágenes

En este capítulo se estudiará la clase **Image** de AWT y el paquete **java.awt.image**. Juntos proporcionan lo necesario para mostrar y manipular *imágenes gráficas*. Una imagen no es más que un objeto gráfico rectangular. Las imágenes son un componente clave del diseño de páginas Web. De hecho, la introducción de la etiqueta **<img>** en el navegador Mosaic de la NCSA (National Center for Supercomputer Applications) fue la causa del tremendo crecimiento de Internet en 1993. Esta etiqueta se utilizaba para incluir una imagen *entre las líneas* del hipertexto. Java amplía este concepto básico y permite gestionar imágenes desde un programa. Debido a la importancia que tienen, Java proporciona un gran soporte para el tratamiento de imágenes.

Las imágenes son objetos de la clase **Image**, que forma parte del paquete **java.awt**. Las imágenes se manipulan utilizando las clases que están en el paquete **java.awt.image**. Hay muchas clases e interfaces para trabajar con imágenes definidas en **java.awt.image** y no será posible estudiarlas todas en este libro. Nos centraremos en las que forman el núcleo básico del tratamiento de imágenes. Las clases de **java.awt.image** que se van a estudiar en este capítulo son las siguientes:

CropImageFilter	MemoryImageSource
FilteredImageSource	PixelGrabber
ImageFilter	RGBImageFilter

Y las interfaces que se utilizarán son:

ImageConsumer	ImageObserver	ImageProducer

También estudiaremos la clase **MediaTracker**, que forma parte de **java.awt**.

Formatos de archivos

Inicialmente, las imágenes de las páginas Web sólo admitían el formato GIF. Este formato fue creado por CompuServe en 1987 para hacer posible que se viesen mientras se cargaban, algo ideal para Internet. Pero las imágenes GIF sólo pueden tener hasta 256 colores, esta limitación hizo que en 1995 la mayoría de los navegadores incorporaran el formato JPEG. El formato JPEG fue creado por expertos en fotografía para almacenar imágenes pudiendo utilizar cualquier color y cualquier

tonalidad. Cuando se crean adecuadamente estas imágenes, pueden tener una fidelidad mucho más alta y estar mucho más comprimidas que su versión en formato GIF. Otro formato disponible es el PNG que también es una alternativa para el formato GIF. En la mayoría de los casos, no hay que preocuparse del formato que se utiliza en los programas. Las clases para manipulación de imagen en Java superan las diferencias entre formatos con una interfaz única.

Conceptos básicos sobre imágenes: creación, carga y visualización

Cuando se trabaja con imágenes, hay tres operaciones habituales: crear una imagen, cargarla y visualizarla. En java, se utiliza la clase **Image** para hacer referencia a las imágenes que están en memoria y a las que se tienen que cargar desde fuentes externas. Java proporciona lo necesario para crear un nuevo objeto imagen, para cargarlo y para visualizarlo. A continuación se verán cada una de estas operaciones.

Creación de un objeto imagen

Tal vez esté pensando que para crear una imagen en memoria hace falta algo como esto:

```
Image test = new Image (200, 100); // Error, esto no trabaja
```

La línea anterior genera un error debido a que las imágenes se tienen que pintar sobre una ventana y la clase **Image** no tiene suficiente información sobre su entorno como para crear el formato de datos adecuados para la pantalla. Por eso, para crear objetos **Image** se utiliza un método llamado **createImage( )** de la clase **Component** de **java.awt**. Recordemos que todos los componentes del AWT son subclases de la clase **Component**, por lo que todos pueden utilizar este método.

El método **createImage( )** proporciona los dos formatos siguientes:

Image createImage(ImageProducer *imgProd*)
Image createImage(int *width*, int *height*)

La primera forma devuelve la imagen producida por *imgProd*, que es un objeto de una clase que implementa la interfaz **ImageProducer** (posteriormente veremos los generadores de imágenes). La segunda forma devuelve una imagen en blanco, es decir, vacía, con la anchura y altura especificadas. A continuación se muestra un ejemplo:

```
Canvas c = new Canvas();
Image test = c.createImage(200, 100);
```

Aquí se crea una instancia de **Canvas** y después se llama al método **createImage( )** para crear realmente un objeto **Image**. En ese instante, la imagen está en blanco. Más tarde se verá cómo escribir datos en ella.

Carga de una imagen

La otra forma de obtener una imagen es cargándola. Para hacerlo, hay que utilizar el método **getImage( )** definido en la clase **Applet**. Tiene los siguientes formatos:

Image getImage(URL *url*)
Image getImage(URL *url*, String *imageName*)

La primera versión devuelve un objeto **Image** que encapsula la imagen que se encuentra en la dirección dada por *url*. La segunda versión devuelve un objeto **Image** que encapsula la imagen que se encuentra en la dirección dada por *url* y que tiene el nombre especificado en *imageName*.

Visualización de una imagen

Una vez que se tiene una imagen, se puede visualizar utilizando **drawImage( )**, que es un método de la clase **Graphics**. Tiene varios formatos, pero aquí sólo usaremos el siguiente:

boolean drawImage(Image *imgObj*, int *left*, int *top*, ImageObserver *imgOb*)

Así se visualiza la imagen que está en *imgObj* con la esquina superior izquierda en la posición que se especifica en *left* y *top*. *imgOb* es una referencia a una clase que implementa la interfaz **ImageObserver**. Todos los componentes AWT implementan esta interfaz. Un observador de imagen es un objeto que puede monitorear una imagen mientras ésta se carga. En la siguiente sección se describe **ImageObserver**.

Con **getImage( )** y **drawImage( )**, es realmente fácil cargar y visualizar una imagen. A continuación se muestra un applet de ejemplo que carga y visualiza una imagen. Se carga el archivo **seattle.jpg**, pero se puede sustituir por cualquier otro archivo GIF o JPG (sólo hay que asegurarse de que esté en el mismo directorio que el archivo HTML que contiene al applet).

```
/*
* <applet code="SimpleImageLoad" width=248 height=146>
*   <param name="img" value="seattle.jpg">
* </applet>
*/
import java.awt.*;
import java.applet.*;

public class SimpleImageLoad extends Applet
{
  Image img;

  public void init() {
    img = getImage (getDocumentBase(), getParameter("img"));
  }

  public void paint(Graphics g) {
    g.drawImage(img, 0, 0, this);
  }
}
```

En el método **init( )**, se asigna a la variable **img** la imagen devuelta por **getImage( )**. El método **getImage( )** utiliza la cadena que devuelve **getParameter("img")** como nombre de archivo para la imagen. Esta imagen se carga desde una **URL** que está relacionada con el resultado de **getDocumentBase( )**, que es la **URL** de la página HTML en la que se encuentra el applet. El nombre de archivo devuelto por **getParameter ("img")** proviene de la etiqueta applet **<param name="img" value="seattle.jpg">.** Es el equivalente, aunque un poco más lento, de utilizar la etiqueta HTML **<img src="seattle.jpg" width=248 height=146>**. En la Figura 25.1 se muestra lo que se ve cuando se ejecuta el programa.

FIGURA 25-1
Ejemplo de la salida de **SimpleImageLoad**

Cuando se ejecuta este applet el método **init( )** carga a **img**. Se puede ir viendo en pantalla cómo se va cargando la imagen desde la red, ya que la implementación en **Applet** de la interfaz **ImageObserver** llama a **paint( )** cada vez que se reciben datos de la imagen.

El hecho de ver cómo se va cargando la imagen da información, pero puede que sea mejor saber en cuánto tiempo se va a cargar para poder hacer otras cosas en paralelo. De esta manera, la imagen completa puede aparecer en la pantalla de una sola vez, cuando esté totalmente cargada. Se puede utilizar **ImageObserver**, que se describe posteriormente, para monitorear o supervisar la carga de una imagen mientras se presenta en pantalla otro tipo de información.

ImageObserver

ImageObserver es un interfaz que se utiliza para recibir el aviso de que se está generando una imagen. **ImageObserver** sólo define un método: **imageUpdate( )**. Utilizar un observador de imágenes permite realizar otras acciones, como mostrar un indicador de progreso u otra pantalla atractiva, mientras se está informado de cómo va la descarga. Este tipo de aviso o notificación es muy útil cuando se está descargando una imagen desde la red, ya que el diseñador de páginas no suele tener en cuenta que hay gente que tiene modems lentos.

El método **imageUpdate( )** tiene el siguiente formato:

boolean imageUpdate(Image *imgObj*, int *flags*, int *izq*, int *sup*, int *ancho*, int *alto*)

Donde *imgObj* es la imagen que se está cargando, y *flags* es un entero que señala el estado del informe de actualización. Los cuatro enteros *izq*, *sup*, *ancho* y *alto* representan un rectángulo que tiene diferentes valores dependiendo de los valores dados a *flags*. **imageUpdate( )** debe devolver **false** si se ha terminado la carga, y **true** si todavía queda imagen por cargar.

El parámetro *flags* contiene uno o más bits definidos como variables estáticas dentro de la interfaz **ImageObserver**. En la Tabla 25-1 se muestran esos indicadores así como la información que proporciona cada uno.

La clase **Applet** tiene una implementación del método **imageUpdate( )** para la interfaz **ImageObserver** que se utiliza para redibujar imágenes mientras estas se van cargando. Se puede sobrescribir este método en nuestra clase para cambiar su comportamiento.

Etiqueta	Significado
WIDTH	El parámetro ancho es válido y contiene la anchura de la imagen.
HEIGHT	El parámetro alto es válido y contiene la altura de la imagen.
PROPERTIES	Las propiedades asociadas a la imagen se pueden obtener utilizando **imgObj.getProperty( ).**
SOMEBITS	Se han recibido más pixeles necesarios para dibujar la imagen. Los parámetros izq, sup, ancho y alto definen el rectángulo que contiene los nuevos pixeles.
FRAMEBITS	Se ha recibido un marco completo que forma parte de una imagen compuesta por múltiples marcos y que previamente ha sido dibujada. No se utilizan los parámetros izq, sup, ancho y alto.
ALLBITS	La imagen está completa. No se utilizan los parámetros izq, sup, ancho y alto.
ERROR	Se ha producido un error cuando se estaba cargando la imagen de forma asíncrona. La imagen está incompleta y no se puede mostrar. No se recibirá información adicional de la imagen. También se activará el indicador **ABORT** para señalar que la generación de la imagen se ha interrumpido.
ABORT	Se ha interrumpido la carga de una imagen que se estaba cargando de forma asíncrona antes de que estuviese completa. Sin embargo, si no se ha producido ningún error, un acceso a cualquier parte de los datos reiniciará la generación de la imagen.

TABLA 25-1 Significado de los bits de bandera del parámetro *flags* en el método **imageUpdate( )**

A continuación se muestra un sencillo ejemplo del método **imageUpdate( )**:

```
public boolean imageUpdate(Image img, int flags,
                           int x, int y, int w, int h) {
  if ((flags & ALLBITS) == 0) {
    System.out.println("La imagen todavía se está procesando.");
    return true;
  } else {
    System.out.println("Terminado el procesamiento de la imagen.");
    return false;
  }
}
```

Doble almacenamiento en búferes

Las imágenes se pueden utilizar no sólo para almacenar dibujos, como se acaba de ver, sino como superficies de dibujo fuera de la pantalla. Eso permite guardar cualquier imagen, incluidos texto y gráficos, en un espacio fuera de la pantalla para visualizarla más adelante. La ventaja es que la imagen se ve sólo cuando está completa. Dibujar una imagen complicada puede llevar varios milisegundos que el usuario percibirá como parpadeos o flashes.

Este parpadeo distrae y hace creer que la presentación es más lenta de lo que es. La utilización de una imagen fuera de pantalla para reducir el parpadeo se conoce como doble almacenamiento en búferes (*double buffering*), ya que se considera a la pantalla como un buffer para píxeles y a la imagen fuera de pantalla como el segundo búfer, donde se pueden preparar los píxeles para visualizarlos.

Al principio de este capítulo, se vio cómo crear un objeto **Image** en blanco. Ahora veremos cómo dibujar en esa imagen en vez de en la pantalla. Recordando capítulos anteriores, se necesita un objeto **Graphics** para utilizar cualquier método de dibujo de Java. Podemos obtener, a través del método **getGraphics( )** el objeto **Graphics** que se va a utilizar para dibujar en un objeto de la clase **Image**. A continuación, se muestra un fragmento de código que crea una nueva imagen, obtiene su contexto gráfico, y rellena la imagen completa con píxeles de color rojo:

```
Canvas c = new Canvas();
Image test = c.createImage(200, 100);
Graphics gc = test.getGraphics();
gc.setColor(Color.red);
gc.fillRect(0, 0, 200, 100);
```

Una vez que se ha construido y rellenado una imagen, todavía no es visible. Para que se vea realmente la imagen, hay que llamar a **drawImage( )**. A continuación se muestra un ejemplo que dibuja una imagen que consume tiempo y permite mostrar las ventajas del doble almacenamiento en búferes:

```
/*
* <applet code=DoubleBuffer width=250 height=250>
* </applet>
*/
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class DoubleBuffer extends Applet {
  int gap = 3;
  int mx, my;
  boolean flicker = true;
  Image buffer = null;
  int w, h;

  public void init() {
    Dimension d = getSize();
    w = d.width;
    h = d.height;
    buffer = createImage(w, h);
    addMouseMotionListener(new MouseMotionAdapter() {
      public void mouseDragged(MouseEvent me) {
        mx = me.getX();
        my = me.getY();
        flicker = false;
        repaint();
      }
      public void mouseMoved(MouseEvent me) {
        mx = me.getX();
        my = me.getY();
```

```
        flicker = true;
        repaint();
      }
    });
  }

  public void paint(Graphics g) {
    Graphics screengc = null;

    if (!flicker) {
      screengc = g;
      g = buffer.getGraphics();
    }

    g.setColor(Color.blue);
    g.fillRect(0, 0, w, h);

    g.setColor(Color.red);
    for (int i=0; i<w; i+=gap)
      g.drawLine(i, 0, w-i, h);
    for (int i=0; i<h; i+=gap)
      g.drawLine(0, i, w, h-i);

    g.setColor(Color.black);
    g.drawString("Pulse el botón del ratón para activar el uso
                 de doble búferes", 10, h/2);

    g.setColor(Color.yellow) ;
    g.fillOval(mx - gap, my - gap, gap*2+1, gap*2+1);

    if (!flicker) {
      screengc.drawImage(buffer, 0, 0, null);
    }
  }
  public void update(Graphics g) {
    paint(g);
  }
}
```

Este sencillo applet tiene un método **paint( )** un poco complicado. Rellena el fondo con color azul y después dibuja encima un patrón en rojo. Sobre esto, se escribe algo de texto en negro y dibuja un círculo amarillo con centro en las coordenadas **mx**, **my**. Se sobrescriben los métodos **mouseMoved( )** y **mouseDragged( )** para obtener la posición del ratón. Estos métodos son idénticos, excepto por el valor que se da a la variable booleana **flicker**. **mouseMoved( )** pone **flicker** a **true**, y **mouseDragged( )** la pone a **false**. Esto hace que se llame a **repaint( )** con **flicker** puesto a **true** cuando se mueve el ratón sin presionar ningún botón, y con **flicker** puesto a **false** cuando se arrastra el ratón con cualquier botón presionado.

Cuando se llama a **paint( )** con **flicker** puesto en **true**, se ve en pantalla cómo se va ejecutando cada una de las operaciones de dibujo. Si se presiona un botón y se llama a **paint( )** con **flicker** puesto a **false**, se ve una imagen bastante diferente. El método **paint( )** cambia la referencia **g** de **Graphics** con el contexto gráfico que hace referencia al **buffer**, fuera de la pantalla, que se ha creado en **init( )**. En este caso, todas las operaciones de dibujo son invisibles y al final del método **paint( )** simplemente se llama a **drawImage( )** para mostrar todo de una sola vez.

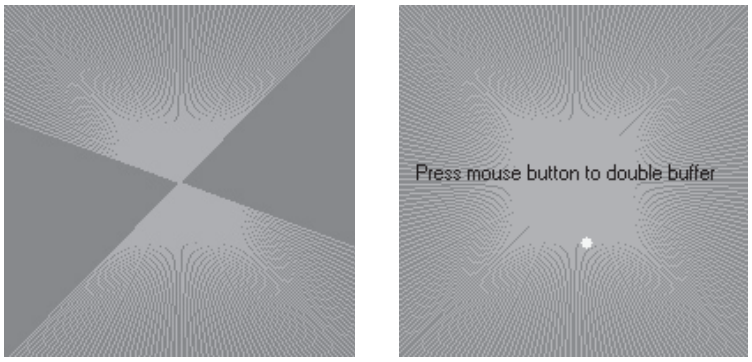


FIGURA 25-2 Salida del applet **DoubleBuffer** sin utilizar y utilizando el doble almacenamiento en búferes

Observe que es correcto pasar **null** como valor del cuarto parámetro de **drawImage( )**. Este es el parámetro que se utiliza para pasar un objeto **ImageObserver** que reciba notificaciones de eventos de la imagen. Debido a que ésta es una imagen que no viene de la red, no es necesario dar ningún aviso o notificación. La imagen izquierda de la Figura 25.2 es la que se genera cuando no se presiona ningún botón del ratón. Como se puede ver, esta imagen se capturó cuando estaba a medio pintar. La imagen de la derecha, generada al presionar un botón del ratón, muestra cómo siempre está completa gracias al doble almacenamiento en búferes.

MediaTracker

Los primeros programadores de Java encontraron a la interfaz **ImageObserver** demasiado difícil de entender y gestionar cuando había muchas imágenes para cargar. Por ese motivo la comunidad de desarrolladores pidió una solución más simple que permitiese a los programadores cargar todas sus imágenes de manera síncrona, sin tener que preocuparse de **imageUpdate( ).** La respuesta vino por parte de Sun Microsystems, que añadió una clase a **java.awt** que se llamó **MediaTracker** en la siguiente versión del JDK. Un **MediaTracker** es un objeto que revisa y comprueba el estado de un número arbitrario de imágenes en paralelo.

Para utilizar **MediaTracker**, se crea una nueva instancia y se utiliza su método **addImage( )** para añadir esa nueva imagen al control de carga. **addImage( )** tiene los siguientes formatos:

void addImage(Image *imgObj*, int *imgID*)
void addImage(Image *imgObj*, int *imgID*, int *width*, int *height*)

Donde *imgObj* es la imagen que se va a controlar. Su número de identificación se pasa en *imgID*. Los números de identificación no tienen qué ser únicos. Se puede utilizar el mismo número con varias imágenes para indicar que pertenecen a un mismo grupo. En el segundo formato, *width* y *height* especifican las dimensiones del objeto cuando se visualiza.

Una vez que se ha registrado una imagen, se puede revisar si está cargada, o esperar a que se cargue completamente. Para revisar el estado de una imagen se llamar a **checkID( )**. La versión que se utiliza en este capítulo es la siguiente:

boolean checkID(int *imgID*)

Donde *imgID* especifica el número de identificación de la imagen que se quiere revisar. El método devuelve **true** si se han cargado todas las imágenes que tienen ese número de identificación (o si se ha terminado la carga por un error o porque ésta ha sido abortada por el usuario).

En caso contrario, devuelve **false**. Se puede utilizar el método **checkAll( )** para ver si se han cargado todas las imágenes controladas.

Se debería utilizar **MediaTracker** cuando se vaya a cargar un grupo de imágenes. Así, si hay alguna imagen de ese grupo que no se descargue, se puede visualizar alguna otra cosa y entretener al usuario mientras llega o llegan las imágenes.

PRECAUCIÓN *Si se utiliza **MediaTracker** una vez que ya se ha llamado a **addImage( )** sobre una imagen, su referencia en **MediaTracker** evitará que el sistema la elimine con el mecanismo de recolección automática de basura. Si se quiere que el sistema pueda mandar a la basura imágenes que se han guardado con dicho mecanismo, hay que asegurarse que la instancia de **MediaTracker** también pueda ser recogida.*

A continuación se presenta un ejemplo que carga un grupo de siete imágenes y muestra una barra que indica cómo avanza la carga:

```
/*
* <applet code="TrackedImageLoad" width=300 height=400>
* <param name="img"
* value="vincent+leonardo+matisse+picasso+renoir+seurat+vermeer">
* </applet>
*/
import java.util.*;
import java.applet.*;
import java.awt.*;

public class TrackedImageLoad extends Applet implements Runnable {
  MediaTracker tracker;
  int tracked;
  int frame_rate = 5;
  int current_img = 0;
  Thread motor;
  static final int MAXIMAGES = 10;
  Image img[] = new Image[MAXIMAGES];
  String name[] = new String[MAXIMAGES];
  boolean stopFlag;

  public void init() {
    tracker = new MediaTracker(this);
    StringTokenizer st = new StringTokenizer(getParameter("img"), "+");

    while(st.hasMoreTokens() && tracked <= MAXIMAGES) {
      name[tracked] = st.nextToken();
      img[tracked] = getImage(getDocumentBase(),
                              name[tracked] + ".jpg");
      tracker.addImage(img[tracked], tracked);
      tracked++;
    }
  }

  public void paint(Graphics g) {
    String loaded = "";
    int donecount = 0;

    for(int i=0; i<tracked; i++) {
```

```
      if (tracker.checkID(i, true)) {
        donecount++;
        loaded += name[i] + " ";
      }
    }

    Dimension d = getSize();
    int w = d.width;
    int h = d.height;

    if (donecount == tracked) {
      frame_rate = 1;
      Image i = img[current_img++];
      int iw = i.getWidth(null);
      int ih = i.getHeight(null);
      g.drawImage(i, (w - iw)/2, (h - ih)/2, null);
    if (current_img >= tracked)
      current_img = 0;
    } else {
      int x = w * donecount / tracked;
      g.setColor(Color.black);
      g.fillRect(0, h/3, x, 16);
      g.setColor(Color.white);
      g.fillRect(x, h/3, w-x, 16);
      g.setColor(Color.black);
      g.drawString(loaded, 10, h/2);
    }
  }

  public void start() {
    motor = new Thread(this);
    stopFlag = false;
    motor.start();
  }

  public void stop() {
    stopFlag = true;
  }

  public void run() {
    motor.setPriority(Thread.MIN_PRIORITY);
    while (true) {
      repaint();
      try {
        Thread.sleep (1000/frame_rate);
      } catch (InterruptedException e) {
        System.out.println("Interrupción");
        return;
      }
      if(stopFlag)
        return;
    }
  }
}
```

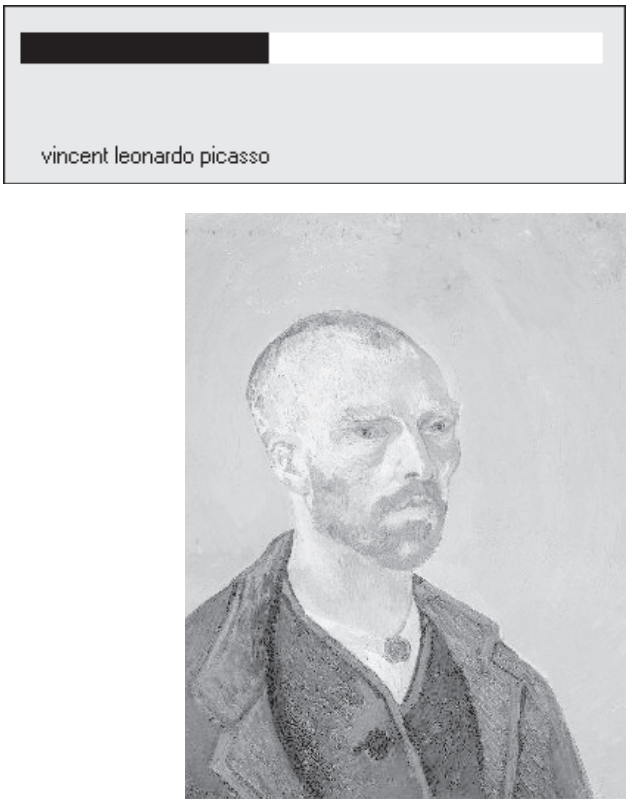


FIGURA 25-3
Ejemplo de la salida de **TrackedImageLoad**

Este ejemplo crea un nuevo objeto **MediaTracker** en el método **init( )**, y después añade cada una de las imágenes con **addImage( )**. En el método **paint( )**, se llama a **checkID( )** para cada una de las imágenes que se están supervisando. Si se han cargado todas las imágenes, se visualizan; si no, se muestra una barra con el número de imágenes cargadas junto con el nombre de esas imágenes debajo de la barra. En la Figura 25.3 se pueden ver dos escenas de la ejecución de este applet. Uno es la barra en la que se dice que se han cargado tres imágenes. El otro es el autorretrato de Van Gogh.

ImageProducer

ImageProducer es una interfaz para objetos que desean generar datos para imágenes. Un objeto que implemente la interfaz **ImageProducer** suministrará arreglos de enteros o de bytes que representan los datos de la imagen y generen objetos **Image**. Como se ha visto anteriormente, uno de los formatos del método **createImage( )** tiene como argumento un objeto **ImageProducer**. En **java.awt.image** hay dos generadores de imágenes: **MemoryImageSource** y **FilteredImageSource**. Aquí examinaremos **MemoryImageSource** y crearemos un nuevo objeto **Image** a partir de los datos generados en un applet.

MemoryImageSource

MemoryImageSource es una clase que crea un nuevo objeto de la clase **Image** a partir de un arreglo de datos. Define varios constructores, de ellos utilizaremos el siguiente:

MemoryImageSource(int *ancho*, int *alto*, int *pixel*[ ], int *offset*, int *anchoLinea*)

El objeto **MemoryImageSource** se construye a partir del arreglo de enteros especificado en *pixel*, y por omisión lo hace en el modelo de colores RGB para producir los datos para un objeto **Image**. En ese modelo de colores por omisión, un píxel es un entero con Alfa, Rojo, Verde y Azul (0xAARRGGBB).El valor Alfa representa el grado de transparencia del píxel. Si Alfa es igual a 0,el píxel es totalmente transparente, y si Alfa es 255, el píxel es totalmente opaco. La anchura y altura de la imagen resultante se pasan en ancho y alto. El punto de partida en el arreglo de píxeles para empezar a leer los datos se pasa en *offset*. La anchura de una línea de escaneado, que suele coincidir con la anchura de la imagen, se pasa en *anchoLinea*.

A continuación se presenta un pequeño ejemplo que genera un objeto **MemoryImageSource** utilizando una variante de un sencillo algoritmo (un OR exclusivo a nivel de bit de los valores x e y de cada píxel) que aparece en el libro *Beyond Photography, The Digital Darkroom*, escrito por Gerard J. Holzmann (Prentice Hall, 1988).

```
/*
* <applet code="MemoryImageGenerator" width=256 height=256>
* </applet>
*/
import java.applet.*;
import java.awt.*;
import java.awt.image.*;

public class MemoryImageGenerator extends Applet {
  Image img;
  public void init() {
    Dimension d = getSize();
    int w = d.width;
    int h = d.height;
    int pixels[] = new int[w * h];
    int i =0;

    for(int y=0; y<h; y++) {
      for(int x=0; x<w; x++) {
        int r = (x^y)&0xff;
        int g = (x*2^y*2)&0xff;
        int b = (x*4^y*4)&0xff;
        pixels[i++]= (255 << 24) | (r << 16) | (g << 8) | b;
      }
    }
    img = createImage(new MemoryImageSource(w, h, pixels, 0, w));
  }
  public void paint(Graphics g) {
    g.drawImage(img, 0, 0, this);
  }
}
```

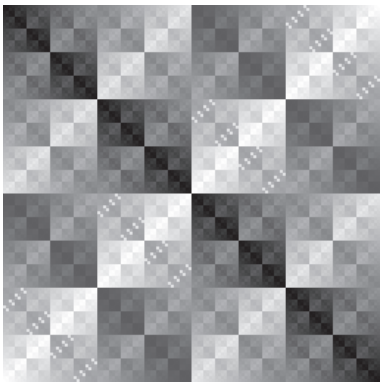


FIGURA 25-4
Ejemplo de la salida de **MemoryImageGenerator**

Los datos del nuevo **MemoryImageSource** se crean en el método **init( )**. Se crea un arreglo de enteros que contiene los valores de los píxeles; los datos se generan en los ciclos **for** anidados, donde los valores **r, g** y **b** pasan a ser un píxel en el arreglo píxeles. Finalmente, se llama a **createImage( )** con una nueva instancia de **MemoryImageSource** creada a partir de los datos de píxel que se le dan como parámetro. En la Figura 25-4 se puede ver la imagen cuando se ejecuta el applet (realmente la imagen se ve mucho mejor con color).

ImageConsumer

ImageConsumer es una interfaz abstracta para objetos que quieren recolectar datos de píxel de imágenes y quieren pasarlos como si fueran otro tipo de datos. Esto es justo lo contrario de **ImageProducer**. Un objeto que implementa la interfaz **ImageConsumer** crea un arreglo de tipo **int** o de tipo **byte** que representan los píxeles de un objeto **Image**. A continuación analizaremos la clase **PixelGrabber**, que es una implementación sencilla de la interfaz **ImageConsumer**.

PixelGrabber

La clase **PixelGrabber** está definida en **java.lang.image** y es la inversa de la clase **MemoryImageSource**. En lugar de construir una imagen a partir de un arreglo de píxel, toma una imagen ya existente y extrae el arreglo de píxeles de ella. Para utilizar **PixelGrabber**, primero se crea un arreglo de enteros que sea suficientemente grande como para poder guardar los datos de píxeles, y después se crea una instancia **PixelGrabber** pasando el rectángulo que se quiere extraer. Finalmente, se llama a **grabPixels( )** en esa instancia.

El constructor **PixelGrabber** que se va a utilizar en este capítulo es el siguiente:

PixelGrabber(Image *imgObj*, int *izq*, int *sup*, int *ancho*, int *alto*, int *pixel*[ ],
int *offset*, int *anchoLinea*)

Aquí, *imgObj* es el objeto del que se van a extraer los píxeles. Los valores de *izq* y *sup* especifican la esquina superior izquierda del rectángulo, y *ancho* y *alto* especifican las dimensiones del rectángulo del que se obtendrán los píxeles. Los píxeles se almacenan en el arreglo *pixel* comenzando en la posición *offset*. La anchura de la línea de escaneado, que suele ser la misma que la anchura de la imagen, se pasa en *anchoLinea*.

El método **grabPixels( )** se define de la siguiente manera:

boolean grabPixels( )
  throws InterruptedException

boolean grabPixels(long *milisegundos*)
  throws InterruptedException

Ambas versiones devuelven **true** si todo ha salido bien y **false** en caso contrario. En el segundo formato, *milisegundos* especifica cuánto tiempo esperará el método los píxeles. Ambos lanzan una excepción **InterruptedException** si la ejecución se interrumpe por otro hilo de ejecución.

A continuación se muestra un ejemplo que extrae los píxeles de una imagen y luego crea un histograma del brillo de los píxeles. El histograma no es más que un conteo de los píxeles que tienen un cierto brillo, comenzando por los que tienen 0 y terminando por los que tienen 255. Después de que el applet pinta la imagen, dibuja el histograma encima.

```
/*
* <appletcode=HistoGrab.class width=341 height=400>
* <param name=img value=vermeer.jpg>
* </applet> */
import java.applet.*;
import java.awt.*;
import java.awt.image.*;

public class HistoGrab extends Applet {
  Dimension d;
  Image img;
  int iw, ih;
  int pixels[];
  int w, h;
  int hist[] = new int[256];
  int max_hist = 0;

  public void init() {
    d = getSize();
    w = d.width;
    h = d.height;

    try {
      img = getImage(getDocumentBase(), getParameter("img"));
      MediaTracker t = new MediaTracker(this);
      t.addImage(img, 0);
      t.waitForID(0);
      iw = img.getWidth(null);
      ih = img.getHeight(null);
      pixels = new int[iw * ih];
      PixelGrabber pg = new PixelGrabber(img, 0, 0, iw, ih,
                                         pixels, 0, iw);

      pg.grabPixels();
    } catch (InterruptedException e) {
      System.out.println("Interrupción");
      return;
    }
```

```
    for (int i=0; i<iw*ih; i++) {
      int p = pixels[i];
      int r = 0xff & (p >> 16);
      int g = 0xff & (p >> 8);
      int b = 0xff & (p);
      int y = (int) (.33 * r + .56 * g + .11 * b);
      hist[y]++;
    }
    for (int i=0; i<256; i++) {
      if (hist[i] > max_hist)
        max_hist = hist[i];
    }
  }

  public void update() {}

  public void paint(Graphics g) {
    g.drawImage(img, 0, 0, null);
    int x = (w - 256) / 2;
    int lasty = h - h * hist[0] / max_hist;
    for (int i=0; i<256; i++, x++) {
      int y = h - h * hist[i] / max_hist;
      g.setColor(new Color(i, i, i));
      g.fillRect(x, y, 1, h);
      g.setColor(Color.red);
      g.drawLine(x-1,lasty,x,y);
      lasty = y;
    }
  }
}
```

En la Figura 25-5 se pueden apreciar la imagen y el histograma de un famoso cuadro de Vermeer.

FIGURA 25-5 Ejemplo de la salida de **HistoGrab**

ImageFilter

Con las dos interfaces **ImageProducer** e **ImageConsumer** —y sus clases concretas **MemoryImageSource** y **PixelGrabber**— se pueden crear un conjunto arbitrario de filtros que toman un conjunto de píxeles, los modifican, y se los pasan a un consumidor arbitrario. Este mecanismo es análogo a aquel en el que se crean clases concretas a partir de las clases abstractas de E/S **InputStream**, **OutputStream**, **Reader** y **Writer** (descritas en el Capítulo 19). Este modelo de flujos para imágenes se completa con la clase **ImageFilter**. Algunas subclases de **ImageFilter** que están en el paquete **java.awt.image** son **AreaAveragingScaleFilter**, **CropImageFilter**, **ReplicateScaleFilter** y **RGBImageFilter**. También hay una implementación de **ImageProducer** llamada **FilteredImageSource**, que toma un arbitrario **ImageFilter** y recubre a un **ImageProducer** para filtrar los píxeles que produce. Se puede utilizar una instancia de **FilteredImageSource** como un **ImageProducer** cuando se llama a **createImage**, de la misma forma que los objetos **BufferedInputStream** se pueden utilizar como objetos **InputStream**.

En este capítulo se analizarán dos de estos filtros: **CropImageFilter** y **RGBImageFilter**.

CropImageFilter

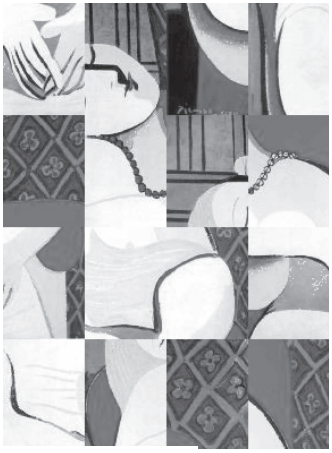
CropImageFilter filtra una imagen origen para extraer una región rectangular. Una de las situaciones en que este filtro es útil es cuando se quiere utilizar varias imágenes pequeñas obtenidas a partir de una sola imagen origen más grande. Lleva mucho más tiempo cargar veinte imágenes de 2K que una sola de 40K. Si todas las subimágenes tienen el mismo tamaño, se podrán extraer fácilmente utilizando **CropImageFilter** para separar el bloque una vez que ha comenzado el applet. A continuación se muestra un ejemplo que crea 16 imágenes a partir de una única imagen. Después se visualizan las 16 imágenes intercambiando 32 veces de forma aleatoria la posición de una pareja de imágenes.

```
/*
 * <applet code=TileImage.class width=288 height=399>
 * <param name=img value=picasso.jpg>
 * </applet>
 */
import java.applet.*;
import java.awt.*;
import java.awt.image.*;

public class TileImage extends Applet {
  Image img;
  Image cell[] = new Image[4*4];
  int iw, ih;
  int tw, th;

  public void init() {
    try {
      img = getImage(getDocumentBase(), getparameter("img"));
      MediaTracker t = new MediaTracker(this);
      t.addImage(img, 0);
      t.waitForID(0) ;
      iw = img.getWidth(null);
      ih = img.getHeight(null);
      tw = iw / 4;
      th = ih / 4;
      CropImageFilter f;
```

```
      FilteredImageSource fis;
      t = new MediaTracker(this);
      for (int y=0; y<4; y++) {
        for (int x=0; x<4; x++) {
          f = new CropImageFilter(tw*x, th*y, tw, th);
          fis = new FilteredImageSource(img.getSource(), f);
          int i = y*4+x;
          cell[i] = createImage(fis);
          t.addImage(cell[i], i);
        }
      }
      t.waitForAll() ;
      for (int i=0; i<32; i++) {
        int si = (int) (Math.random() * 16);
        int di = (int) (Math.random() * 16);
        Image tmp = cell[si];
        cell[si] = cell[di];
        cell [di] = tmp:
      }
    } catch (InterruptedException e) {
      System.out.println("Interrupción");
    }
  }

  public void update(Graphics g) {
    paint(g) :
  }

  public void paint(Graphics g) {
    for (int y=0; y<4; y++) {
      for (int x=0; x<4: x++) {
        g.drawImage(cell[y*4+x], x * tw, y * th, null);
      }
    }
  }
}
```

En la Figura 25-6 se muestra un famoso cuadro de Picasso transformado con el applet **TileImage**.

FIGURA 25-6 Ejemplo de la salida **TileImage**



RGBImageFilter

El **RGBImageFilter** se utiliza para convertir una imagen en otra, píxel a píxel, transformando los colores. Se podría utilizar este filtro para dar brillo a una imagen, para incrementar su contraste, o para pasarla a escala de grises.

Para explicar **RGBImageFilter**, se ha desarrollado un ejemplo un poco complicado, que emplea una estrategia para conectar dinámicamente los filtros que procesan las imágenes. Se ha creado una interfaz para filtrar imágenes, por lo que el applet puede simplemente cargar esos filtros basado en etiquetas **<param>** sin tener que conocer los filtros por adelantado. El ejemplo está formado por una clase principal llamada **ImageFilterDemo**, la interfaz llamada **PlugInFilter**, y una clase llamada **LoadedImage**, que encapsula alguno de los métodos **MediaTracker** que se han utilizado en este capítulo. También se incluyen tres filtros —**Grayscale**, **Invert**, y **Contrast**— que manipulan el color de la imagen original utilizando **RGBImageFilters**, y dos clases más —**Blur** y **Sharpen**— que realizan los filtros de "convolución", los cuales cambian los datos de píxel basándose en los píxeles que rodean a cada uno de los píxeles originales. **Blur** y **Sharpen** son subclases de una clase auxiliar llamada **Convolver**. Veamos cada una de las partes del ejemplo.

ImageFilterDemo.java

La clase **ImageFilterDemo** es el entorno de trabajo para este ejemplo de filtros de imágenes. Utiliza un **BorderLayout**, con un **Panel** en la posición *South* para poner los botones que van a representar a cada uno de los filtros. La posición *North* la ocupa un objeto **Label** para mostrar mensajes de información sobre el funcionamiento de los filtros. La imagen se ubica en la posición *Center* y se encapsula en la subclase **LoadedImageCanvas**, descrita más adelante. Los botones / filtros de la etiqueta <**param**> de nombre **filters** se obtienen utilizando un **StringTokenizer** para analizar el valor del parámetro (los filtros están separados con signos +).

El método **actionPerformed( )** es interesante ya que utiliza la etiqueta de un botón como el nombre de la clase de filtro que trata de cargar con (**PlugInFilter**) **Class.forName(a). newInstance( ).** Este método es robusto y toma una acción adecuada si el botón no corresponde a una clase que implemente **PlugInFilter**.

```
/*
* <applet code=ImageFilterDemo width=350 height=450>
* <param name=img value=vincent.jpg>
* <param name=filters value="Grayscale+Invert+Contrast+Blur+ Sharpen">
* </applet>
*/
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class ImageFilterDemo extends Applet implements ActionListener {
  Image img;
  PlugInFilter pif;
  Image fimg;
  Image curImg;
  LoadedImage lim;
  Label lab;
  Button reset;
```

```
  public void init() {
    setLayout(new BorderLayout());
    Panel p = new panel();
    add(p, BorderLayout.SOUTH);
    reset = new Button("Reset");
    reset.addActionListener(this);
    p.add(reset);
    StringTokenizer st = new StringTokenizer(getParameter("filters"), "+");
    while(st.hasMoreTokens()) {
      Button b = new Button(st.nextToken());
      b.addActionListener(this);
      p.add(b);
  }

    lab = new Label("");
    add(lab, BorderLayout.NORTH);

    img = getImage(getDocumentBase(), getParameter("img"));
    lim = new LoadedImage(img);
    add(lim, BorderLayout.CENTER);

}

public void actionPerformed(ActionEvent ae) {
  String a = "";

  try{
    a = ae.getActionCommand();
    if (a.equals("Reset")) {
      lim.set(img);
      lab.setText("Normal");
    }
    else{
      pif = (PlugInFilter) Class.forName(a).newInstance();
      fimg = pif.filter(this, img);
      lim.set(fimg);
      lab.setText("Filtro: " + a);
    }
    repaint();
  } catch (ClassNotFoundException e) {
    lab.setText(a + " no encontrada");.
    lim.set(img);
    repaint();
    } catch (InstantiationException e) {
      lab.setText("No se pudo crear un nuevo " + a);
    } catch (IllegalAccessException e) {
      lab.setText("Sin acceso a: " + a);
    }
  }
}
```

La Figura 25-7 muestra el aspecto del applet cuando se carga por primera vez utilizando la etiqueta applet que aparece al comienzo del código fuente.

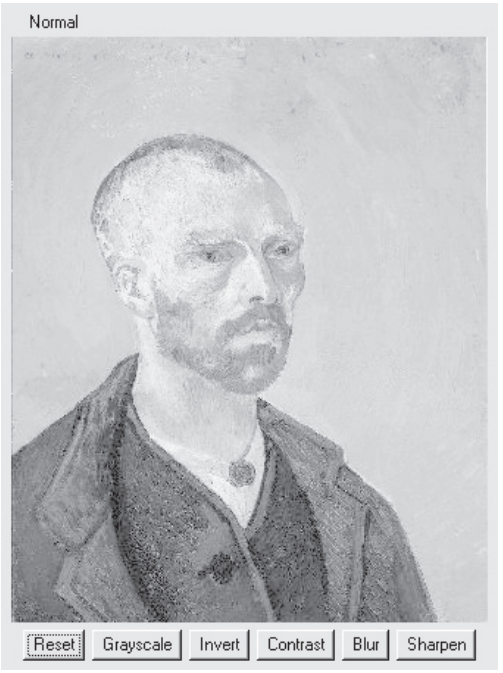


FIGURA 25-7 Ejemplo de la salida de **ImageFilterDemo**

PlugInFilter.java

PlugInFilter es una interfaz sencilla que se utiliza para filtrar imágenes abstractas. Sólo tiene un método, **filter( )**, que a partir del applet y de la imagen original devuelve una nueva imagen que ha sufrido algún tipo de filtrado.

```
interface PlugInFilter {
  java.awt.Image filter(java.applet.Applet a, java.awt.Image in);
}
```

LoadedImage.java

LoadedImage es una subclase de **Canvas**, que toma una imagen en tiempo de construcción, y a la vez la va cargando utilizando **MediaTracker**. **LoadedImage** se comporta de manera adecuada dentro del control **LayoutManager**, ya que sobrescribe los métodos **getPreferredSize( )** y **getMinimumSize( ).** También tiene un método llamado **set( )** que se puede utilizar para establecer una nueva **Image** y mostrarla en ese **Canvas**. Así es como se visualiza la imagen filtrada después de que finalice la carga.

```
import java.awt.*;

public class LoadedImage extends Canvas {
  Image img;

  public LoadedImage{Image i) {
    set(i);
  }

  void set(Image i) {
    MediaTracker mt = new MediaTracker(this);
```

```
    mt.addImage(i, 0);
    try {
      mt.waitForAll() ;
    } catch (InterruptedException e) {
      System.out.println("Interrupción");
      return;
    }
    img = i;
    repaint();
  }

  public void paint(Graphics g) {
    if (img == null) {
      g.drawString("sin imagen", 10, 30);
    } else {
      g.drawImage(img, 0, 0, this);
    }
  }

    public Dimension getPreferredSize() {
      return new Dimension(img.getWidth(this), img.getHeight(this));
    }

    public Dimension getMinimumSize() {
      return getPreferredSize();
    }
  }
```


Grayscale.java

El filtro **Grayscale** es una subclase de **RGBImageFilter**, lo que significa que se puede utilizar el propio **Grayscale** como el parámetro **ImageFilter** del constructor **FilteredImageSource**. Lo único que hay que hacer para cambiar los valores de los colores de entrada es sobrescribir **filterRGB( ).** Este método toma los valores de los colores rojo, verde y azul y calcula el brillo del píxel, utilizando el factor de conversión de color a brillo del NTSC (National Television Standards Committee). Después, simplemente devuelve un píxel gris que tiene el mismo brillo que el píxel en color original.

```
import java.applet.*;
import java.awt.*;
import java.awt.image.*;

class Grayscale extends RGBImageFilter implements PluglnFilter {
  public Image filter(Applet a, Image in) {
    return a.createImage(new FilteredImageSource(in.getSource(), this));
  }

  public int filterRGB(int x, int y, int rgb) {
    int r = (rgb >> 16) & 0xff;
    int g = (rgb >> 8) & 0xff;
    int b = rgb & 0xff;
    int k = (int) (.56 * g + .33 * r + .11 * b);
    return (0xff000000 | k << 16 | k << 8 | k);
  }
}
```

Invert.java

El filtro **Invert** también es bastante sencillo. Toma los colores rojo, verde y azul, y los invierte restándolos a 255. Estos valores invertidos se juntan en un píxel y se devuelve el valor de ese píxel.

```
import java.applet.*;
import java.awt.*;
import java.awt.image.*;

class Invert extends RGBImageFilter implements PluglnFilter {
  public Image filter(Applet a, Image in) {
    return a.createImage(new FilteredImageSource(in.getSource(), this));
  }
  public int filterRGB(int x, int y, int rgb) {
    int r = 0xff - (rgb >> 16) & 0xff;
    int g = 0xff - (rgb >> 8) & 0xff;
    int b = 0xff - rgb & 0xff;
    return (0xff000000 | r << 16 | g << 8 | b);
  }
}
```

En la Figura 25-8 se muestra la imagen después de haber ejecutado el filtro **Invert**.

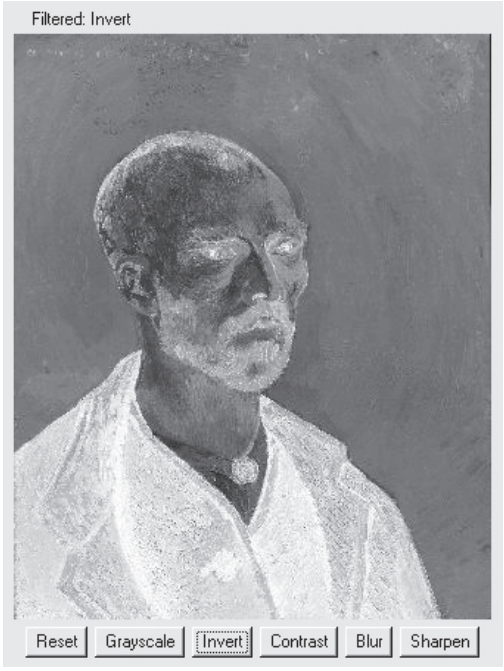


FIGURA 25-8
Uso del filtro **Invert** con **ImageFilterDemo**

Contrast.java

El filtro **Contrast** es muy similar a **Grayscale**, excepto que sobrescribe **filterRGB( )** de una manera un poco más complicada. El algoritmo que utiliza para mejorar el contraste toma por separado los valores del rojo, verde y azul y los multiplica por 1.2 si su brillo es mayor de 128. Si es menor de 128, se dividen entre 1.2. Con el método **multclamp( )**, se evita que los valores que se multipliquen por 1.2 sobrepasen el valor de 255.

```
import java.applet.*;
import java.awt.*;
import java.awt.image.*;

public class Contrast extends RGBImageFilter implements PluglnFilter {

  public Image filter(Applet a, Image in) {
    return a.createImage(new FilteredImageSource(in.getSource(), this));
  }

  private int multclamp(int in, double factor) {
    in = (int) (in * factor);
    return in > 255 ? 255 : in;
  }

  double gain = 1.2;
  private int cont(int in) {
    return (in < 128) ? (int) (in/gain) : multclamp(in, gain);
  }

  public int filterRGB(int x, int y, int rgb) {
    int r = cont((rgb >> 16) & 0xff);
    int g = cont((rgb >> 8) & 0xff);
    int b = cont(rgb & 0xff);
    return (0xff000000 | r << 16 | g << 8 | b);
  }
}
```

En la Figura 25-9 se muestra la imagen después de presionar el botón **Contrast**.

Convolver.java

La clase abstracta **Convolver** gestiona los fundamentos de un filtro de convolución implementando la interfaz **ImageConsumer** para mover los píxeles originales a un arreglo que se llama **imgpixels**. También crea un segundo arreglo llamado **newimgpixels** para los datos filtrados. En el ejemplo se filtra un pequeño rectángulo de píxeles alrededor de cada píxel de una imagen, que se llama *núcleo de convolución*. Esta área, que en la demo es de 3 × 3 píxeles, se utiliza para decidir cómo cambiar el píxel central en esa área.

---

*NOTA La razón por la que el filtro no puede modificar directamente el arreglo **imgpixels** es que el siguiente píxel en una línea de escaneado trataría de utilizar el valor original del píxel anterior, que ya habría sido filtrado.*

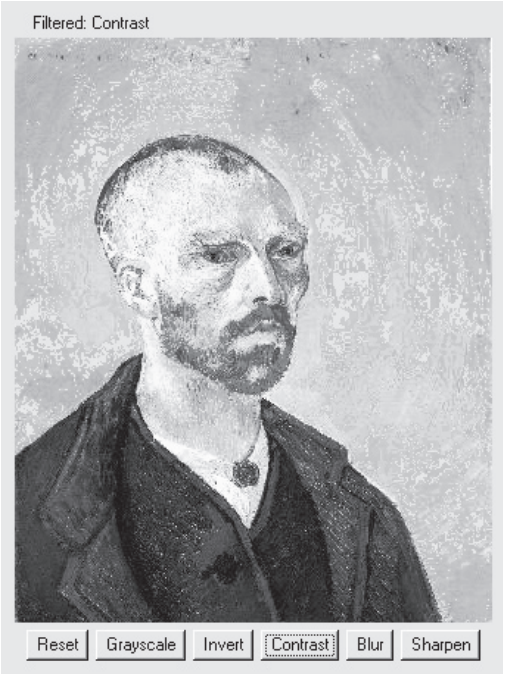


FIGURA 25-9
Uso del filtro **Contrast** con **ImageFilterDemo**

Las dos subclases concretas que aparecen en la siguiente sección simplemente implementan el método **convolve( )**, utilizando **imgpixels** para los datos originales y **newimgpixels** para guardar el resultado.

```
import java.applet.*;
import java.awt.*;
import java.awt.image.*;

abstract class Convolver implements ImageConsumer, PluglnFilter {
  int width, height;
  int imgpixels[], newimgpixels[];
  boolean imageReady = false;

  abstract void convolve(); // aquí va el filtro...

  public Image filter(Applet a, Image in) {
    in.getSource().startProduction(this);
    imageReady = false;
    waitForImage();
    newimgpixels = new int[width*height];

    try{
      convolve();
    } catch (Exception e) {
      System.out.println("Fallo en la convolución: " + e);
      e.printStackTrace();
    }
```

```
  return a.createImage(
    new MemoryImageSource(width, height, newimgpixels, 0, width));
}

synchronized void waitForImage() {
  try {
    while (!imageReady) wait();
  } catch (Exception e) {
    System.out.println("Interrupción");
  }
}

public void setProperties(java.util.Hashtable dummy) { }
public void setColorModel(ColorModel dummy) { }
public void setHints(int dummy) { }

public synchronized void imageComplete(int dummy) {
imageReady = true;
  notifyAll() ;
}

public void setDimensions(int x, int y) {
  width = x;
  height = y;
  imgpixels = new int [x*y] ;
}

public void setPixels(int xl, int yl, int w, int h,
  ColorModel model, byte pixels[], int off, int scansize) {
  int pix, x, y, x2, y2, sx, sy;

  x2 = xl+w;
  y2 = yl+h;
  sy = off;
  for(y=yl; y<y2; y++) {
    sx = sy;
    for(x=xl; x<x2; x++) {
      pix = model.getRGB(pixels[sx++]);
      if((pix & 0xff000000) == 0)
          pix = 0x00ffffff;
      imgpixels[y*width+x] =pix;
    }
    sy += scansize;
  }
}

public void setPixels(int xl, int yl, int w, int h,
  ColorModel model, int pixels[], int off, int scansize) {
  int pix, x, y, x2, y2, sx, sy;

  x2 = xl+w;
  y2 = yl+h;
  sy = off;
  for(y=yl; y<y2; y++) {
```

```
      sx = sy;
      for(x=xl; x<x2; x++) {
        pix = model.getRGB(pixels[sx++]);
        if((pix & 0xff000000) == 0)
            pix = 0x00ffffff;
        imgpixels[y*width+x] = pix;
      }
      sy += scansize;
    }
  }
}
```

Blur.java

El filtro **Blur** es una subclase de **Convolver** y no hace más que recorrer todos los píxeles del arreglo de la imagen original, **imgpixels**, y calcular la media del rectángulo de 3 × 3 que hay alrededor de cada píxel. El píxel de salida correspondiente se guarda en **newimgpixels**, y es precisamente ese valor medio.

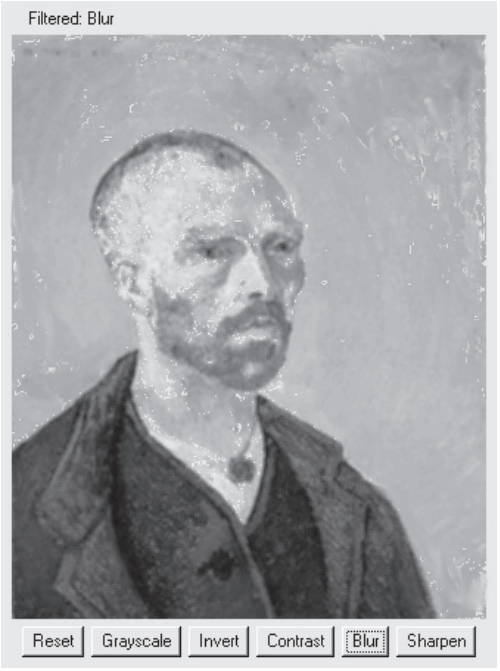
```
public class Blur extends Convolver {
  public void convolve() {
    for(int y=1; y<height-1; y++) {
      for(int x=1; x<width-1; x++) {
        int rs = 0;
        int gs = 0;
        int bs = 0;

        for(int k=-1; k<=1; k++) {
          for(int j=-1; j<=1; j++) {
            int rgb = imgpixels[(y+k)*width+x+j];
            int r = (rgb >> 16) & 0xff;
            int g = (rgb >> 8) & 0xff;
            int b = rgb & 0xff;
            rs += r;
            gs += g;
            bs += b;
          }
        }

        rs /= 9;
        gs /= 9;
        bs /= 9;

      new imgpixels[y*width+x] = (0xff000000 |
                                  rs << 16 | gs << 8 | bs);
      }
    }
  }
}
```

En la Figura 25-10 se muestra el applet después de ejecutar **Blur**.

Sharpen.java

El filtro **Sharpen** también es una subclase de **Convolver**, y más o menos es el inverso de **Blur**. Recorre todos los píxeles en el arreglo de la imagen original, **imgpixels**, y calcula la media del área de 3 × 3 que rodea cada píxel, pero sin contar el centro.

FIGURA 25-10
Uso del filtro **Blur** con **ImageFilteDemo**



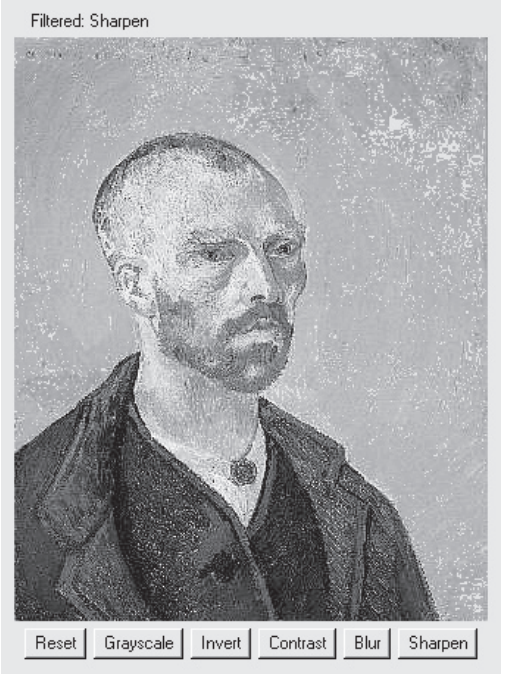
El correspondiente píxel de salida, que se guarda en **newimgpixels**, tiene como valor el valor del píxel central más la diferencia que haya entre éste y la media de lo que le rodea; es decir, que si un píxel tiene un valor de 30 (en la escala de 0 a 255) más brillante que los píxeles de su alrededor, el filtro le añade otros 30 de brillo; sin embargo, si tiene un valor de 10 más oscuro, lo hace otros 10 más oscuro. Esto tiende a acentuar los bordes, mientras que deja sin cambios las zonas homogéneas.

```
public class Sharpen extends Convolver {

  private final int clamp(int c) {
    return (c > 255 ? 255 : (e < 0 ? 0 : c));
  }

  public void convolve() {
    int r0=0, g0=0, b0=0;
    for(int y=1; y<height-1; y++) {
      for(int x=1; x<width-1; x++) {
        int rs = 0;
        int gs = 0;
        int bs = 0;

        for(int k=-1; k<=1; k++) {
          for(int j=-1; j<=1; j++) {
            int rgb = imgpixels[(y+k)*width+x+j];
            int r = (rgb >> 16) & 0xff;
            int g = (rgb >> 8) & 0xff;
            int b = rgb & 0xff;
            if (j == 0 && k == 0) {
              r0 = r;
```

```
                g0 = g;
                b0 = b;
              } else {
                rs += r;
                gs += g;
                bs += b;
              }
            }
          }

          rs >>= 3;
          gs >>= 3;
          bs >>= 3;
          newimgpixels[y*width+x] = (0xff000000 |
                                     clamp(r0+r0-rs) << 16 |
                                     clamp(g0+g0-gs) << 8 |
                                     clamp(b0+b0-bs));
        }
      }
    }
  }
```

En la Figura 25-11 se muestra el applet después de haber ejecutado **Sharpen**.

FIGURA 25-11
Uso del filtro **Sharpen** con **ImageFilterDemo**



Animación de imágenes

Una vez que se ha visto de forma general las APIs de imágenes, podemos juntar lo visto en un interesante applet que va a visualizar una secuencia de animación de imágenes. Las imágenes para la animación se toman de una única imagen que se puede dividir en una malla, cuyo tamaño vienen dado por las etiquetas <**param**> **rows** y **cols**. Cada celda de esa malla de imágenes se obtiene de forma similar a la utilizada en el ejemplo **TileImage**. La secuencia en que se van a visualizar esa celdas se determina con la etiqueta <**param**> **sequence**. Este parámetro es una lista de los números de las celdas, separados por comas, iniciando con el valor cero, de izquierda a derecha y de arriba abajo.

Una vez que se han analizado las etiquetas <**param**> del applet y se ha cargado la imagen original, se divide la imagen en un cierto número de subimágenes. Después, se comienzan a visualizar las imágenes en el orden que aparezca en **sequence**. En la secuencia de imágenes, entre una y otra pasa el tiempo suficiente para poder verlas. El código fuente es el siguiente:


```
// Ejemplo de animación.
import java.applet.*;
import java.awt.*;
import java.awt.image.*;
import java.util.*;

public class Animation extends Applet implements Runnable {
  Image cell[] ;
  final int MAXSEQ = 64;
  int sequence[];
  int nseq;
  int idx;
  int framerate;
  boolean stopFlag;

  private int intDef(String s, int def) {
    int n = def;
    if (s != null)
      try {
        n = Integer.parselnt(s);
      } catch (NumberFormatException e) {
        System.out.println("Excepción en el Formato de Números");
      }
    return n;
  }

  public void init() {
    framerate = intDef(getParameter("framerate"), 5);
    int tilex = intDef (getParameter ("cols"), 1);
    int tiley = intDef (getParameter ("rows"), 1);
    cell = new Image[tilex*tiley];

    StringTokenizer st = new
              StringTokenizer(getParameter("sequence"), ",");
    sequence = new int[MAXSEQ];
```

```
  nseq = 0;
  while(st.hasMoreTokens() && nseq < MAXSEQ) {
    sequence[nseq] = intDef(st.nextToken(), 0);
    nseq++;
  }

  try {
    Image img = getImage(getDocumentBase(), getParameter("img"));
    MediaTracker t = new MediaTracker(this) ;
    t.addImage (img, 0);
    t.waitForID(0) ;
    int iw = img.getWidth(null);
    int ih = img.getHeight(null);
    int tw = iw / tilex;
    int th = ih / ti1ey;
    CropImageFilter f;
    FilteredImageSource fis;
    for (int y=0; y<tiley; y++) {
      for (int x=0; x<tilex; x++) {
        f = new CropImageFilter(tw*x, th*y, tw, th);
        fis = new FilteredImageSource(img.getSource(), f);
        int i = y*tilex+x;
        cell[i] = createImage(fis);
        t.addImage(cell[i], i);
      }
    }
    t. waitForAll() ;
  } catch (InterruptedException e) {
    System.out.println("Carga de la Imagen Interrumpida");
  }
}

public void update(Graphics g) { }

pub1ic void paint(Graphics g) {
    g.drawImage(cell[sequence[idx]], 0, 0, null);
}

Thread t;
pub1ic void start() {
  t = new Thread(this);
  stopF1ag = fa1se;
  t.start();
}

pub1ic void stop(){
  stopF1ag = true;
}

pub1ic void run() {
```

```
      idx = 0;
      while (true) {
        paint(getGraphics());
        idx = (idx + 1) % nseq;
        try {
          Thread.s1eep(1000/framerate);
        } catch (InterruptedException e) {
          System.out.println("Animación Interrumpida");
        return;
      }
      if(stopF1ag)
        return;
    }
  }
}
```

La siguiente etiqueta de applet muestra el famoso estudio del movimiento de caballos de Eadweard Muybridge, que demuestra que hay algún instante en que los caballos no tienen ninguna pata apoyada en el suelo. Por supuesto, se puede sustituir el archivo de imagen por otro cualquiera.

```
<app1et code=Animation width=67 height=48>
<param name=img value=horse.gif>
<param name=rows va1ue=3>
<param name=co1s va1ue=4>
<param name=sequence va1ue=0,1,2,3,4,5,6,7,8,9,10,11>
<param name=framerate va1ue=l5>
</app1et>
```

En la Figura 25-12 se muestra la ejecución del applet. Obsérvese que se ha cargado la imagen original debajo del applet utilizando una etiqueta <**img**> normal.

FIGURA 25-12 Ejemplo de la salida de **Animation**

Más clases para trabajo con imágenes

Además de las clases de imágenes que se han descrito en este capítulo, **java.awt.image** proporciona muchas otras que posibilitan el control sobre el proceso de imágenes y que admiten técnicas gráficas avanzadas. Está disponible además el paquete **javax.imageio** para el trabajo con imágenes. Este paquete soporta diferentes formatos de imágenes. Si se está interesado en salidas con gráficos sofisticados, encontrará muy interesante explorar el resto de las clases contenidas en el paquete **java.awt.image** y **javax.imageio.**

26 CAPÍTULO

Utilerías para concurrencia

Desde el inicio Java ha proporcionado soporte para programación multihilos y sincronización. Por ejemplo, es posible crear hilos implementando la interfaz **Runnable** o heredando de la clase **Thread**, la sincronización está disponible mediante el uso de la palabra reservada **synchronized**, y la comunicación entre hilos está soportada por los métodos **wait( )** y **notify( )**, mismos que están definidos en la clase **Object**. En general, el soporte para programación multihilos de Java fue una de las innovaciones más importantes de Java y aún una de sus principales fortalezas.

Sin embargo, la pureza conceptual del soporte original de Java para programación multihilos no es ideal para todas las aplicaciones – especialmente en aquellas que hacen extenso uso de hilos. Por ejemplo, el soporte original para programación multihilo no proporciona diversas características de alto nivel, tales como semáforos, grupo de hilos, y administradores de ejecución, que facilitan la creación de programas concurrentes intensivos.

Para comenzar es importante explicar que muchos programas en Java hacen uso de múltiples hilos y son por lo tanto"concurrentes". Por ejemplo, la mayoría de los applets utilizan múltiples hilos. Sin embargo, el término concurrente se utiliza en este capítulo para referirnos a un programa que hace uso integral y extensivo de hilos de ejecución que trabajan de manera concurrente. Un ejemplo de este tipo de programa es uno que utiliza hilos separados para calcular simultáneamente el resultado parcial de un cálculo grande. Otro ejemplo es un programa que coordina las actividades de varios hilos, cada una de las cuales busca accesar a la información de una base de datos. En este caso, el acceso de sólo lectura podría ser manejado de una forma diferente que en aquellos en los que se requiera acceso con capacidades de lectura y escritura.

Para gestionar las necesidades de un programa concurrente, JDK 5 agregó las utilerías para concurrencia, conocidas comúnmente como el API de concurrencia. Las utilerías para concurrencia suministran muchas características que han sido requeridas durante mucho tiempo por los programadores que desarrollan aplicaciones concurrentes. Por ejemplo, estas utilerías ofrecen semáforos, barreras, contadores de bloqueo, grupos de hilos, administradores de ejecución, candados, varias colecciones concurrentes, y una forma estilizada de utilizar hilos para obtener resultados computacionales.

El API de concurrencia es bastante grande, y muchas de las cuestiones alrededor de su uso son bastante complejas. Está fuera del alcance de este libro discutir todas sus facetas. Además la alternativa ofrecida por las utilerías de concurrencia no está diseñada para ser utilizada en la mayoría de los programas. En pocas palabras, a menos que esté escribiendo un programa con una cantidad significativa de concurrencia, en la mayoría de los casos, el soporte tradicional de Java para multihilos y sincronización es suficiente y en muchos casos preferible a las capacidades ofrecidas por el API de concurrencia.

A pesar del párrafo anterior es importante que todos los programadores cuenten con un conocimiento general de cómo funciona el API de concurrencia.

Además, hay algunas partes del API, tales como los sincronizadores, los hilos bajo demanda, y los ejecutores, que se pueden aplicar a una amplia variedad de situaciones. Por esas razones, este capítulo presenta una visión general de las utilerías de concurrencia y muestra varios ejemplos de su uso.

El API para trabajo con concurrencia

Las utilerías de concurrencia están contenidas en el paquete **java.util.concurrent** y en sus dos subpaquetes **java.util.concurrent.atomic** y **java.util.concurrent.locks**. A continuación damos una visión general de sus contenidos.

java.util.concurrent

java.util.concurrent define las características principales que soportan alternativas para las estrategias predefinidas de sincronización y comunicación entre hilos. El paquete define las siguientes características clave:

Sincronizadores
Ejecutores
Colecciones concurrentes

Los sincronizadores ofrecen formas de sincronización de alto nivel para la interacción entre múltiples hilos. Las clases sincronizadoras definidas por **java.util.concurrent** son:

Semaphore	Implementa al semáforo clásico.
CountDownLatch	Espera hasta que un número especificado de eventos hayan ocurrido.
CyclicBarrier	Habilita a un grupo de hilos para esperar en un punto predefinido de ejecución.
Exchanger	Intercambia datos entre dos hilos.

Note que cada sincronizador provee una solución a un problema específico de sincronización. Esto permite que cada sincronizador sea optimizado para uso específico. En el pasado, este tipo de sincronizaciones se hacían a mano. El API de concurrencia los estandariza y los pone a disposición para todos los programadores de Java.

Los ejecutores administran la ejecución de hilos. A la cabeza de la jerarquía de ejecutores está la interfaz **Executor**, la cual es utilizada para iniciar un hilo. **ExecutorService** extiende a **Executor** y provee métodos que gestionan la ejecución. Existen dos implementaciones de **ExecutorService**: **ThreadPoolExecutor** y **ScheduledThreadPoolExecutor**. **java**.**util**. **concurrent** también define la clase **Executors**, la cual incluye varios métodos estáticos que simplifican la creación de ejecutores.

Relacionadas con los ejecutores están las interfaces **Future** y **Callable**. Un objeto de tipo **Future** contiene un valor que es devuelto por un hilo después de su ejecución. Así, su valor se convierte por definición "en el futuro"cuando el hilo termine. **Callable** define un hilo que devuelve un valor.

java.util.concurrent define diversas clases de colecciones concurrentes, incluyendo **ConcurrentHashMap**, **ConcurrentLinkedQueue** y **CopyOnWriteArraylist**. Éstas ofrecen alternativas concurrentes para las clases equivalentes definidas por la estructura de colecciones.

Finalmente, para gestionar mejor el tiempo de un hilo, **java.util.concurrent** define la enumeración **TimeUnit**.

java.util.concurrent.atomic

java.util.concurrent.atomic facilita el uso de variables en un ambiente concurrente. Provee una forma eficiente de actualizar el valor de una variable sin el uso de candados. Esto se logra a través del uso de clases, tales como **AtomicInteger** y **AtomicLong**, y métodos como, **compareAndSet( )**, **decrementAndGet( )** y **getAndSet( )**. Estos métodos se ejecutan como una operación única e ininterrumpible.

java.util.concurrent.locks

java.util.concurrent.locks proporciona una alternativa al uso de métodos de sincronizados. El núcleo de esta alternativa es la interfaz **Lock**, la cual define el mecanismo básico utilizado para adquirir y liberar el acceso a un objeto. Los métodos clave son **lock( ), tryLock( )** y **unlock( )**. La ventaja de utilizar estos métodos es el mayor control sobre la sincronización.

El resto de este capítulo analiza más de cerca de los componentes del API de concurrencia.

Uso de objetos para sincronización

Probablemente la parte más ampliamente utilizada del API de concurrencia serían los objetos de sincronización. Los objetos de sincronización están soportados por las clases **Semaphore**, **CountDownLatch**, **CyclicBarrier** y **Exchanger**. En conjunto, estas clases permiten gestionar con facilidad situaciones de sincronización anteriormente complicadas. Además son aplicables a un amplio rango de programas, incluso aquellos que contienen sólo una limitada concurrencia. Debido a que seguramente los objetos de sincronización serán de interés para todos los programas en Java, cada uno es examinado con detalle aquí.

Semaphore

El objeto de sincronización que muchos lectores reconocerán inmediatamente es el objeto de tipo **Semaphore**, el cual implementa un semáforo clásico. Un semáforo controla el acceso a un recurso compartido a través del uso de un contador. Si el contador es mayor que 0, entonces el acceso es *permitido*. Si el contador es cero, entonces el acceso es denegado. Lo que el contador está contando son los permisos que autorizan el acceso al recurso compartido. De esta forma, para acceder a un recurso, un hilo debe recibir un pase de acceso desde el semáforo.

En general para utilizar un semáforo, el hilo que quiere acceder al recurso compartido intenta adquirir el permiso. Si el contador del semáforo es mayor que cero, entonces el hilo adquiere un permiso lo cual causa que el contador del semáforo sea decrementado. De otra forma, el hilo será bloqueado hasta que un permiso pueda ser adquirido. Cuando el hilo no necesita más el acceso al recurso compartido, libera el permiso, lo cual causa que el contador del semáforo sea incrementado. Si existe otro hilo esperando por un permiso, entonces ese hilo adquirirá un permiso en ese momento. La clase **Semaphore** de Java implementa este mecanismo.

La clase **Semaphore** tiene los dos constructores que se muestran a continuación:

Semaphore(int *num*)
Semaphore(int *num*, boolean *p*)

Donde, *num* especifica el valor inicial del contador. Esto es, *num* especifica el número de hilos que pueden acceder a un recurso compartido al mismo tiempo. Si *num* es uno, entonces sólo un hilo puede acceder al recurso al mismo tiempo.

Por omisión, se asigna el permiso a los hilos que esperan por él en un orden indefinido. Establecer el valor *p* a **verdadero**, asegura que el permiso sea asignado en el orden en el que los hilos solicitaron el acceso.

Para adquirir el permiso, se llama al método **acquire( )**, el cual tiene estas dos formas:

void adquire( ) throws InterruptedException
void adquire(int *num*) throws InterruptedException

La primera forma adquiere un permiso. La segunda forma adquiere el número de permisos definidos en *num*. Regularmente se utiliza la primera forma. Si el permiso no puede ser obtenido al tiempo de la llamada, entonces el hilo que invoca se suspende hasta que el permiso le sea otorgado.

Para liberar el permiso, se llama al método **release( )**, el cual tiene estas dos formas:

void release( )
void release(int *num*)

La primera forma libera un permiso. La segunda forma libera el número de permisos especificados en *num*.

Para utilizar un semáforo como control de acceso a un recurso, cada hilo que quiere usar el recurso debe primero llamar al método **acquire( )** antes de acceder al recurso. Cuando el hilo ha terminado de utilizar el recurso, debe llamar al método **release( )**. A continuación hay un ejemplo que ilustra el uso de un semáforo.

```
// Un ejemplo simple de objetos de tipo Semaphore

import java.util.concurrent.*;

class SemDemo {

  public static void main(String args[]) {
    Semaphore sem = new Semaphore(1);

    new IncThread(sem, "A");
    new DecThread(sem, "B");

  }
}

// Un recurso compartido
class Shared {
  static int count = 0;
}

// Un hilo de ejecución que incrementa count.
class IncThread implements Runnable {
  String name;
  Semaphore sem;

  IncThread(Semaphore s, String n) {
    sem = s;
    name = n;
    new Thread (this).start();
  }
```

```
  public void run() {
    System.out.println ("Comenzando " + name);
    try {
      // Primero, obtiene el permiso
      System.out.println(name + " está esperando por permiso")
      sem.acquire();
      System.out.println(name + " obtiene permiso.");
      // Ahora, accede al recurso compartido
      for (int i=0; i < 5; i++) {
        Shared.count++;
        System.out.println(name + ": " + Shared.count);
        // Ahora, permite un cambio de contexto - si es posible.
        Thread.sleep(10);
      }
    } catch (InterruptedException exc) {
      System.out.println(exc);
    }
    // Libera el permiso
    System.out.println (name + " libera el permiso.");
    sem.release();
  }
}
// Un hilo de ejecución que decrementa count.
class DecThread implements Runnable {
  String name;
  Semaphore sem;
  DecThread (Semaphore s, String n) {
    sem = s;
    name = n;
    new Thread(this).start();
  }
  public void run () {
    System.out.println("Comenzando " + name);
    try {
      // Primero obtiene el permiso.
      System.out.println(name + " está esperando permiso.")
      sem.acquire ();
      System.out.println(name + " obtiene permiso");
      // Ahora, accede al recurso compartido
      for (int i=0; i < 5; i++) {
        Shared.count--;
        System.out.println (name + ": " + Shared.count);
```

```
      // Ahora, permite un cambio de contexto - si es posible
      Thread.sleep(10);
      }
    } catch (InterruptedException exc) {
      System.out.println(exc);
    }

    // Libera el permiso
    System.out.println(name + " libera el permiso.")
    sem.release();
  }
}
```

La salida del programa se muestra aquí. El orden preciso en el cual los hilos se ejecutan podría variar.

```
Comenzando A
A está esperando permiso
A obtiene permiso.
A: 1
Comenzando B
B está esperando permiso
A: 2
A: 3
A: 4
A: 5
A libera el permiso.
B obtiene permiso.
B: 4
B: 3
B: 2
B: 1
B: 0
B libera el permiso
```

El programa utiliza un semáforo para controlar el acceso a la variable **count**, la cual es una variable estática dentro la clase **Shared**. La variable **Shared.count** se incrementada cinco veces en el método **run( )** de **IncThread** y decrementada cinco veces en **DecThread**. Para prevenir que esos dos hilos accedan a **Shared.count** al mismo tiempo, el acceso está permitido sólo después de adquirir un permiso desde el semáforo de control. Después de que el acceso está completo, el permiso se libera. En esta forma, sólo un hilo accederá a **Shared.count**, como se muestra en la salida.

Tanto en **IncThread** como en **DecThread**, nótese la llamada al método **sleep( )** dentro del método **run( ).** Esto se utiliza para"probar"que el acceso a **Shared.count** está sincronizado con el semáforo. En el método **run( )**, la llamada a **sleep( )** causa que el hilo que invoca haga pausa entre cada acceso a **Shared.count**. Esto normalmente habilitaría al segundo hilo. Sin embargo, debido al semáforo, el segundo hilo debe esperar hasta que el primero libere el permiso, lo cual ocurre sólo después de que todos los accesos del primer hilo se completan. De esta forma, **Shared.count** primero es incrementado cinco veces en **IncThread** y luego decrementado cinco veces por **DecThread**. Los incrementos y decrementos no se mezclan.

Sin el uso del semáforo, el acceso a **Shared**.**count** por ambos hilos hubiera ocurrido simultáneamente, y los incrementos y decrementos hubieran sido mezclados. Para confirmar esto, intente comentar las llamadas a los métodos **acquire( )** y **release( )**. Cuando se ejecute el programa, se verá que el acceso a la variable **Shared**.**count** ya no está sincronizado, y cada hilo accede a ella en cuanto obtiene un intervalo de tiempo.

Aunque muchos usos de un semáforo son tan directos como se ha mostrado en el programa anterior, otros usos más fascinantes también son posibles. A continuación un ejemplo. El siguiente programa reconstruye el programa del productor/consumidor mostrado en el Capítulo 11 utilizando dos semáforos para regular a los hilos productor y consumidor, asegurando que cada llamada al método **put( )** es seguida por la correspondiente llamada al método **get( )**:


```
// Una implementación de un productor y consumidor
// que utiliza semáforos para controlar la sincronización

import java.util.concurrent.Semaphore;

class Q {
  int n;

  // Comienza con el semáforo no disponible para el consumidor
  static Semaphore semCon = new Semaphore(0);
  static Semaphore semProd = new Semaphore(1);

  void get() {
    try {
      semCon.acquire();
    } catch (InterruptedException e) {
      System.out.println (" Se generó una InterruptedException");
    }

    System.out.println("Obtiene : " + n);
    semProd. release ();
  }

  void put(int n) {
    try {
      semProd.acquire();
    } catch( InterruptedException e) {
      System.out.println("Se generó un InterruptedException");
    }

    this.n = n;
    System.out.println("Pone : " + n);
    semCon.release ();
  }
}

class Producer implements Runnable {
  Q q;

  Producer(Q q) {
    this.q = q;
```

```
    new Thread(this,"Productor").start()
  }

  public void run() {
    for(int i=0; i < 20; i++) q.put(i);
  }
}

class Consumer implements Runnable {
  Q q;

  Consumer(Q q) {
    this.q = q;
    new Thread(this, "Consumidor").start();
  }

  public void run(){
    for(int i=0; i < 20; i++) q.get();
  }
}

class ProdCon {
  public static void main(String args[]) {
    Q q = new Q();
    new Consumer(q);
    new Producer(q);
  }
}
```

Una porción de la salida del programa se muestra a continuación:

```
Pone     0
Obtiene  0
Pone     1
Obtiene  1
Pone     2
Obtiene  2
Pone     3
Obtiene  3
Pone     4
Obtiene  4
Pone     5
Obtiene  5
.
.
.
```

Como puede ver, las llamadas a **put( )** y **get( )** están sincronizadas. Esto es, cada llamada a **put( )** es seguida por un llamada a **get( )** y ningún valor se pierde. Sin los semáforos, múltiples llamadas a **put( )** habrían ocurrido sin la correspondiente llamada a **get( )**, con la correspondiente perdida de valores. Para probar esto, remueva el código del semáforo y observe el resultado.

La secuencia de llamadas a los métodos **put( )** y **get( )** es gestionada por dos semáforos: **semProd** y **semCon**. Antes de que **put( )** pueda producir un valor, debe adquirir el permiso del semáforo **semProd**. Después de que se ha fijado el valor, libera a **semCon**. Antes de que **get( )** pueda consumir un valor, debe adquirir el permiso de **Semyon**. Después de consumido el valor, libera a **semProd**. Este mecanismo de "dar y tomar" asegura que cada llamada al método **put( )** debe estar seguida por una llamada al método **get( )**.

Note que **semCon** es inicializado con permisos no disponibles. Esto asegura que **put( )** se ejecute primero. La habilidad de definir el estado de sincronización inicial es uno de los más poderosos aspectos de un semáforo.

CountDownLatch

Algunas veces será deseable que un hilo espere hasta que uno o más eventos hayan ocurrido. Para manejar este tipo de situaciones, el API de concurrencia proporciona al **CountDownLatch**. Un **CountDownLatch** es creado inicialmente con un contador del número de eventos que deben ocurrir antes de que el bloqueo se libere. Cada vez que un evento ocurre, el contador es decrementado. Cuando el contador alcanza cero, el bloqueo se abre.

CountDownLatch tiene el siguiente constructor:

CountDownLatch (int *num*)

Donde, *num* especifica el número de eventos que deben ocurrir para que el bloqueo se abra.

Para esperar a un bloqueo, un hilo llama al método **await( )**, el cual tiene la forma que se muestra a continuación:

void await( ) throws InterruptedException
void await(long *esp*, TimeUnit *tu*) throws InterruptedException

La primera forma espera hasta que el contador asociado con el **CountDownLatch** que invoca alcanza cero. En la segunda forma espera solo por el periodo de tiempo especificado en el argumento llamado *esp*. Las unidades representadas por *esp* son especificadas por *tu*, el cual es un objeto de tipo enumeración **TimeUnit**. **TimeUnit** es descrito posteriormente en este capítulo.

Una señal de un evento, llama al método **countDown( ),** como se muestra aquí:

void countDown( )

Cada llamada a **countDown( )** decrementa al contador asociado con el objeto que invoca.

El siguiente programa muestra el uso de **CountDownLatch**. Crea un bloqueo que requiere que ocurran cinco eventos antes de que se abra.

```
//Un ejemplo de CountDownLatch

import java.util.concurrent.CountDownLatch;

class CDLDemo {
  public static void main(String args[]) {
    CoundDownLatch cdl = new CountDownLatch(5);

    System.out.println("Comenzando");

    new MyThread(cdl);
```

```
    try {
      cdl.await();
    } catch (InterruptedException exc) {
      System.out.println(exc);
    }

    System.out.println("Hecho");
  }
}

class MyThread implements Runnable {
  CountDownLatch latch;

  MyThread(CountDownLatch c) {
    latch = c;
    new Thread(this).start();
  }

  public void run() {
    for(int i = 0 ; i<5; i++) {
      System.out.println (i);
      latch.countDown(); // decrementa count
    }
  }
}
```

La salida producida por el programa se muestra a continuación:

```
Comenzando
0
1
2
3
4
Hecho
```

Dentro de **main( )** se crea un **CountDownLatch** llamado **cdl** con un contador inicial de cinco. A continuación, se crea una instancia de **MyThread**, la cual comienza la ejecución de un nuevo hilo. Note que **cdl** se pasa como parámetro al constructor de **MyThread** y almacenado en la variable de instancia **latch**. Luego, el hilo principal llama al método **await( )** del objeto **cdl**, el cuál causa que la ejecución del hilo principal se detenga hasta que el contador de **cdl** sea decrementado cinco veces.

Dentro del método **run( )** de **MyThread**, se crea un ciclo que itera cinco veces. En cada iteración, el método **countDown( )** es llamado para **latch**, la cual se refiere a **cdl** en el método **main( )**. Después de la quinta iteración, el bloqueo se abre, lo cual permite al hilo principal reactivarse.

CountDownLatch es un objeto poderoso para sincronización, fácil de usar y apropiado siempre que un hilo deba esperar la ocurrencia de uno o más eventos.

CyclicBarrier

Una situación poco común en programación concurrente ocurre cuando un grupo de dos o más hilos deben esperar en un predeterminado punto de ejecución hasta que todos los hilos en el grupo hayan alcanzado ese punto.

Para gestionar tal situación, el API de concurrencia provee la clase **CyclicBarrier**. La cual permite la definición de un objeto de sincronización que se suspende hasta que el número especificado de hilos haya alcanzado el punto establecido.

CyclicBarrier tiene los siguientes dos constructores:

CyclicBarrier(int *numHilos*)
CyclicBarrier(int *numHilos*, Runnable *ac*)

Donde, *numHilos* especifica el número de hilos que deben alcanzar la barrera antes de que la ejecución continúe. En la segunda forma, **ac** especifica a un hilo que será ejecutado cuando la barrera sea alcanzada.

Aquí está el procedimiento general que se debe seguir para utilizar a **CyclicBarrier**. Primero, se crea un objeto **CyclicBarrier**, especificando el número de hilos que se estará esperando. A continuación, cuando cada hilo alcance el límite, se llama al método **await( )**. Esto detiene la ejecución del hilo hasta que todos los otros hilos también llamen al método **await( )**. Una vez que el número especificado de hilos haya alcanzado el límite, **await( )** regresará, y la ejecución continuará. Además, si se ha especificado una acción en el parámetro, dicho hilo será ejecutado.

El método **await( )** tiene las siguientes dos formas:

int await( ) throws InterruptedException, BrokenBarrierException

int await(long *esp*, TimeUnit *tu*)
  throws InterruptedException, BrokenBarrierException, TimeoutException

La primera forma espera hasta que todos los hilos han alcanzado el punto límite. La segunda forma espera sólo por el periodo de tiempo especificado por *esp*. Las unidades representadas por *esp* están especificadas por *tu*. Ambas formas regresan un valor que indica el orden en que los hilos llegan al punto límite. El primer hilo devuelve un valor igual al número de hilos a ser atendidos menos uno. El último hilo devuelve cero.

Aquí hay un ejemplo que muestra el funcionamiento de **CyclicBarrier**. El ejemplo espera hasta que un grupo de tres hilos han alcanzado el límite. Cuando eso ocurre, el hilo especificado por **BarAction** se ejecuta.

```
// Ejemplo de CyclicBarrier.

import.java.util.concurrent.*;

class BarDemo {
  public static void main(String args[]) {
    CyclicBarrier cb = new CyclicBarrier(3, new BarAction());

    System.out.println("Comenzando");

    new MyThread(cb, "A");
    new MyThread(cb, "B");
    new MyThread(cb, "C");

  }
}

//Un hilo de ejecución que utiliza un CyclicBarrier
```

```
class MyThread implements Runnable {
  CyclicBarrier cbar;
  String name;

  MyThread (CyclicBarrier c, String n) {
    cbar = c;
    name = n;
    new Thread(this).start();
  }

  public void run() {

    System.out.println(name);

    try {
      cbar await();
    } catch (BrokenBarrierException exc) {
      System.out.println(exc);
    } catch (InterruptedException exc) {
      System.out.println(exc);
    }
  }
}

// Un objeto de esta clase es llamada cuando el
// CyclicBarrier termina.
class BarAction implements Runnable {
  public void run() {
    System.out.println("Límite Alcanzado.");
  }
}
```

La salida se muestra a continuación. El orden exacto en el cual los hilos se ejecutan puede variar.

```
Comenzando
A
B
C
Límite Alcanzado
```

Un **CyclicBarrier** puede ser reutilizado porque será liberada esperando hilos cada vez que el número especificado de hilos llame a **await( )**. Por ejemplo, si se cambia a **main( )** en el programa anterior, de la siguiente forma:

```
public static void main(String args[]) {
CyclicBarrier cb = new CyclicBarrier(3, new BarAction());

  System.out.println("Comenzando");

  new MyThread(cb, "A");
  new MyThread(cb, "B");
  new MyThread(cb, "C");
```

```
  new MyThread(cb, "X");
  new MyThread(cb, "Y");
  new MyThread(cb, "Z");
}
```

La siguiente salida será generada. El orden exacto en el cual los hilos se ejecuten podría variar.

```
Comenzando
A
B
C
Límite alcanzado
X
Y
Z
Límite alcanzado
```

Como lo muestra el ejemplo anterior, **CyclicBarrier** ofrece una solución estilizada de lo que antes fue un problema complicado.

Exchanger

Posiblemente la más interesante de las clases de la sincronización es la clase **Exchanger**. Esta clase está diseñada para simplificar el intercambio de datos entre dos hilos. El funcionamiento de un **Exchanger** es asombrosamente simple: simplemente espera hasta que dos hilos separados llamen al método **exchange( )**. Cuando eso ocurre, se lleva a cabo el intercambio de datos proporcionados por los hilos. Este mecanismo es tanto elegante como fácil de usar. Los usos de **Exchanger** son simples de imaginar. Por ejemplo, un hilo puede preparar un bufer para recibir información desde una conexión de red. Otro hilo llenaría dicho bufer con la información obtenida de la conexión. Los dos hilos trabajarían en conjunto para que cada vez que se necesitase un nuevo bufer, se realice un intercambio.

Exchanger es una clase genérica que está declarada como se muestra aquí:

Exchanger <V>

Donde, **V** especifica el tipo de los datos que están siendo intercambiados.

El único método definido por **Exchanger** es **exchange( )**, el cual tiene las dos formas siguientes:

V exchange(V *bufer*) throws InterruptedException

V exchange(V *bufer*, long *esp*, TimeUnit *tu*)
  throws InterruptedException, TimeoutException

Donde, *bufer* es una referencia a los datos a ser intercambiados. Los datos recibidos desde el otro hilo son devueltos. La segunda forma de **exchange( )** permite especificar un periodo de tiempo de desconexión. El punto clave del método **exchange( )** es que no terminará hasta que haya sido llamado para el mismo objeto **Exchanger** por dos hilos independientes. De esa forma, el método **exchange( )** sincroniza el intercambio de datos.

En el siguiente ejemplo se muestra el uso de **Exchanger**. El ejemplo crea dos hilos. En un hilo se crea un buffer vacío que recibirá los datos que entregará el segundo hilo. De esta forma, el primer hilo intercambia un bufer vacío por uno lleno.

```
// Ejemplo de Exchanger.

import java.util.concurrent.Exchanger;

class ExgrDemo {
  public static void main(String args[] ) {
    Exchanger<String> exgr = new Exchanger<String> () ;

    new UseString(exgr);
    new MakeString(exgr);
  }
}

// Un hilo que construye una cadena.
class MakeString implements Runnable {
  Exchanger<String> ex;
  String str;

  MakeString (Exchanger<String> c) {
    ex = c;
    str = new String();

    new Thread (this).start();
  }

  public void run() {
    char ch = 'A' ;

    for (int i = 0; i < 3 ; i ++) {

      // Llena el Bufer
      for (int j = 0; j < 5; j++)
        str += ch++;

      try {
        // Intercambia un bufer lleno por uno vacío.
        str = ex. exchange(str);
      } catch(InterruptedException exc) {
        System.out.println(exc);
      }
    }
  }
}

// Un hilo que utiliza una cadena
class UseString implements Runnable {
Exchanger<String> ex;
String str;

  UseString(Exchanger <String> c) {
    ex = c;
    new Thread(this).start();
  }
```

```
  public void run() {
    for (int i = 0; i < 3; i++) {
      try {
        //Intercambia un bufer vacío por uno lleno
        str = ex.exchange(new String());
        System.out.println("Obtiene:" +str);
      } catch(InterruptedException exc) {
        System.out.println(exc);
      }
    }
  }
}
```

A continuación se muestra la salida del programa:

```
Obtiene: ABCDE
Obtiene: FGHIJ
Obtiene: KLMNO
```

En el programa, el método **main( )** crea un objeto **Exchanger** para cadenas. Este objeto es utilizado para sincronizar el intercambio de cadenas entre las clases **MakeString** y **UseString**. La clase **MakeString** llena una cadena con datos. La clase **UseString** intercambia un bufer vacío por uno lleno. Luego despliega el contenido de la nueva cadena. El intercambio de bufers vacíos y llenos está sincronizado por el método **exchange( )**, el cual es llamado por el método **run( )** de ambas clases.

Uso de Executor

El API de concurrencia proporciona lo que se denomina"un *ejecutor*", el cual inicializa y controla la ejecución hilos. Como tal, un ejecutor ofrece una alternativa para la gestión de hilos diferente a la clase **Thread**.

El núcleo de un ejecutor es la interfaz **Executor**. La cual define el siguiente método:

void execute(Runnable *h*)

El hilo especificado por el argumento *h* es ejecutado. De esta forma, el método **execute( )** ejecuta al hilo especificado.

La interfaz **ExecutorService** extiende de **Executor** y agrega métodos que ayudan a manejar y controlar la ejecución de hilos. Por ejemplo, **ExecutorService** define al método **shutdown( )**, mostrado a continuación, el cual detiene al **ExecutorService** que invoca.

void shutdown( )

ExecutorService también define métodos que ejecutan hilos que devuelven resultados, métodos que ejecuta un grupo de hilos y métodos que determinan el estado de finalización.Veremos varios de estos métodos más adelante.

También está definida la interfaz **ShceduledExecutorService**, la cual extiende a **ExecutorService** para soportar la planificación de hilos.

El API de concurrencia define dos clases de ejecutores: **ThreadPoolExecutor** y **SchedulerThreadPoolExcecutor**. **ThreadPoolExecutor** implementa a las interfaces **Executor** y **ExecutorService** y proporciona soporte para administrar un conjunto de hilos. **SchedulerThreadPoolExcecutor** también implementa a la interfaz **ScheduledExcecutorService** para permitir la planificación de un conjunto de hilos.

Un conjunto de hilos es un grupo de hilos utilizado para ejecutar varias tareas. En lugar de que cada tarea tenga su propio hilo, se utilizan los hilos en el conjunto. Esto reduce la sobrecarga asociada con la creación de varios hilos separados. Aunque se puede utilizar directamente a **ThreadPoolExecutor** y a **SheduledThreadPoolExecutor**, comúnmente se obtiene un ejecutor llamando uno de los siguientes métodos estáticos de fábrica definidos por la clase de utilería **Executor**.

static ExecutorService newCachedThreadPool( )
static ExecutorService newFixedThreadPool(int *numHilos*)
static ScheduledExecutorService newScheduledThreadPool(int *numHilos*)

El método **newCachedThreadPool** crea un conjunto de hilos que añade hilos conforme son requeridos y reutiliza hilos siempre que es posible. El método **newFixedThreadPool** crea un conjunto de hilos que consiste en un número específico de hilos. El método **newScheduledThreadPool** crea un conjunto de hilos que permite planeación con los hilos. Cada método devuelve una referencia a un **ExecutorService** que puede ser empleado para administrar al conjunto.

Un ejemplo simple de Executor

Antes de continuar, es momento de revisar un ejemplo sencillo con un ejecutor. El siguiente programa crea un grupo de hilos que contiene dos hilos. El programa utiliza el grupo para ejecutar cuatro tareas. De esta forma, cuatro tareas comparten los dos hilos que están en el grupo. Después de que las tareas son realizadas, el grupo se cierra y el programa termina.

```
// Un ejemplo sencillo que utilice un objeto Executor.

import java.útil.concurrent .*;

class SimpExec {
  public static void main(String args[]) {
    CountDownLatch cdl = new CountDownLatch(5);
    CountDownLatch cdl2 = new CountDownLatch(5);
    CountDownLatch cdl3 = new CountDownLatch(5);
    CountDownLatch cdl4 = new CountDownLatch(5);
    ExecutorService es = Executors.newFixedThreadPool(2);

    System.out.println("Comenzando");

    // Comienza el hilo.
    es.execute(new MyThread (cdl, "A"));
    es.execute(new MyThread(cdl2, "B"));
    es.execute(new MyThread(cdl3, "C"));
    es.execute(new MyThread(cdl4, "D"));

    try {
      cdl.await();
```

```
      cdl2.await();
      cdl3.await();
      cdl4.await();
    } catch (InterruptedException exc) {
      System.out.println(exc);
    }

    es.shutdown();
    System.out.println("Hecho");
  }
}

class MyThread implements Runnable {
  String name;
  CountDownLatch latch;

  MyThread(CountDownLatch c, String n) {
    latch = c;
    name = n;

    new Thread(this);
  }

  public void run() {

    for(int i = 0; i < 5; i++) {
      System.out.println(name + ": " + i)
      latch.countDown();
    }
  }
}
```

La salida del programa se muestra a continuación. El orden exacto en el cual los hilos se ejecutan podría variar.

```
Comenzando
A: 0
A: 1
A: 2
A: 3
A: 4
C: 0
C: 1
C: 2
C: 3
C: 4
D: 0
D: 1
D: 2
D: 3
D: 4
B: 0
B: 1
B: 2
B: 3
B: 4
Hecho
```

Como la salida muestra, aunque el grupo de hilos contiene solamente dos hilos, las cuatro tareas son ejecutadas. Sin embargo, sólo dos pueden ejecutarse al mismo tiempo. Las otras dos deben esperar hasta que alguno de los hilos esté disponible para su uso.

La llamada al método **shutdown( )** es importante. Si no estuviera presente en el programa, el programa no terminaría debido a que el ejecutor permanecería activo. Para verificar esto, solamente tenemos que comentar la llamada a **shutdown( )** y observar el resultado.

Uso de Callable y Future

Una de las características más innovadoras y emocionantes del API de concurrencia es la nueva interfaz **Callable**. Esta interfaz representa a un hilo que devuelve un valor. Una aplicación puede utilizar objetos **Callable** para calcular resultados que son devueltos al hilo que invoca. Éste es un mecanismo poderoso porque facilita la codificación de muchos tipos de cálculos numéricos en los cuales se calculan resultados parciales simultáneamente. También se pueden utilizar para ejecutar un hilo que devuelve un código de estado que indica que el hilo ha sido completado satisfactoriamente.

Callable es una interfaz genérica que está definida como sigue:

```
interface Callable<V>
```

Donde, **V** representa el tipo de datos devueltos por la tarea. **Callable** define sólo un método llamado **call( )**, el cual se muestra a continuación:

```
V call( ) throws Exception
```

Dentro de **call( )**, se define la tarea que será ejecutada. Después de que la tarea sea completada, se devolverá el resultado. Si el resultado no puede ser calculado **call( )** debe generar una excepción.

Una tarea definida en un objeto **Callable** es ejecutada por un **ExecutorService**, llamando a su método **submit( )**. Existen tres formas del método **submit( )**, pero sólo una es utilizada para ejecutar a **Callable**. El método se muestra a continuación:

```
<T> Future <T> submit (Callable<T> tarea)
```

Donde, *tarea* es el objeto **Callable** que será ejecutado en su propio hilo. El resultado se devolverá a través de un objeto de tipo **Future**.

Future es una interfaz genérica que representa el valor que será devuelto por un objeto de tipo **Callable**. Debido a que este valor se obtiene en algún momento futuro, el nombre **Future**, para la interfaz, resulta más que apropiado. La interfaz **Future** está definida como:

```
interface Future<V>
```

Donde, **V** especifica el tipo del resultado.

Para obtener el valor devuelto, se llamará al método **get( )** de la interfaz **Future**, el cual tiene estas dos formas:

```
V get( )
  throws InterruptedException, ExecutionException

V get(long esp, TimeUnit tu)
  throws InterruptedException, ExecutionException, TimeoutException
```

La primera forma espera el resultado indefinidamente. En la segunda forma se permite especificar un tiempo de espera en *esp*. Las unidades de *esp* se pasan en el argumento *tu*, el cual es un objeto de la enumeración **TimeUnit**, que se describirá más adelante en este capítulo.

El siguiente programa ejemplifica el uso de las interfaces **Callable** y **Future** creando tres tareas que ejecutan tres diferentes cálculos. La primera tarea devuelve la suma de un valor, la segunda calcula el valor de la hipotenusa de un triángulo dada la longitud de sus catetos, y la tercera calcula el factorial de un valor. Los tres cálculos ocurren simultáneamente.


```
// Un ejemplo que utiliza la interfaz Callable.

import java.util.concurrent.*;

class CallableDemo {
  public static void main(String args[]) {
    ExecutorService es = Executors.newFixedThreadPool(3);
    Future<Integer> f;
    Future<Double> f2;
    Future<Integer> f3;

    System.out.println("Comenzando");

    f = es.submit(new Sum(10));
    f2 = es.submit(new Hypot(3, 4));
    f3 = es.submit(new Factorial(5));

    try {
      System.out.println(f.get());
      System.out.println(f2.get());
      System.out.println(f3.get());
    } catch (InterruptedException exc) {
      System.out.println(exc);
    }
    catch (ExecutionException exc) {
      System.out.println(exc);
    }

    es.shutdown();
    System.out.println("Hecho");
  }
}

// A continuación están los tres hilos

class Sum implements Callable<Integer> {
  int stop;

  Sum(int v) { stop = v; }

  public Integer call() {
    int sum = 0 ;
    for (int i = 1; i <= stop; i++){
      sum += i;
    }
    return sum;
  }
}
```

```
class Hypot implements Callable<Double> {
  double sidel, side2;

  Hypot (double si, double s2) {
    sidel = si;
    side2 = s2;
  }

  public Double call() {
    return Math.sqrt ((sidel*sidel) + (side2*side2));
  }
}

class Factorial implements Callable<Integer> {
  int stop;

  Factorial (int v) { stop = v; }

  public Integer call() {
    int fact = 1;
    for(int i = 2; i <= stop; i++) {
      fact *= i;
    }
    return fact;
  }
}
```

La salida se muestra aquí:

```
Comenzando
55
5.0
120
Hecho
```

La enumeración de tipo TimeUnit

El API de concurrencia define varios métodos que toman un argumento de tipo **TimeUnit**, el cual indica un tiempo de espera. **TimeUnit** es una enumeración que se utiliza para especificar la *granularidad* (o resolución) del tiempo. **TimeUnit** está definido dentro **java.util.concurrent**. Puede ser uno de los siguientes valores:

DAYS
HOURS
MINUTES
SECONDS
MICROSECONDS
MILLISECONDS
NANOSECONDS

Los primeros tres fueron agregados por Java SE 6.

Aunque **TimeUnit** permite especificar cualquiera de estos valores en llamadas a métodos que toman un argumento de tiempo, no existe garantía de que el sistema sea capaz de tomar la resolución especificada.

A continuación hay un ejemplo que utiliza **TimeUnit**. La clase **CallableDemo**, mostrada en la sección previa, es modificada como se presenta a continuación para utilizar la segunda forma de **get( )** y tomar un argumento **TimeUnit**.

```
try {
  System.out.println(f.get(10,TimeUnit.MILLISECONDS));
  System.out.println(f2.get(10,TimeUnit.MILLISECONDS));
  System.out.println(f3.get(10,TimeUnit.MILLISECONDS));
} catch (InterruptedException exc) {
  System.out.println(exc);

}
catch (ExecutionException exc) {
  System.out.println(exc);
} catch (TimeoutException exc) {
  System.out.println(exc);
}
```

En esta versión, ninguna llamada a **get( )** tomará más de 10 milisegundos.

La enumeración **TimeUnit** define varios métodos que realizan conversiones entre unidades. Estos se muestran a continuación:

long convert(long *tval*, TimeUnit *tu*)
long toMicros(long *tval*)
long toMillis(long *tval*)
long toNanos(long *tval*)
long toSeconds(long *tval*)
long toDays(long *tval*)
long toHours(long *tval*)
long toMinutes (long *tval*)

El método **convert( )** convierte el valor *tval* en la unidad especificada y devuelve el resultado. Los métodos ejecutan la conversión indicada y devuelven el resultado. Los últimos tres métodos fueron agregados por Java SE 6.

TimeUnit también define los siguientes métodos de tiempo:

void sleep(long *delay*) throws InterruptedException
void timedJoin(Thread *thrd*, long *delay*) throws InterruptedException
void timedWait(Object *obj*, long *delay*) throws InterruptedException

Donde **sleep( )** detiene la ejecución durante un periodo específico de tiempo, el cual está definido en términos de la constante de enumeración que invoca. Se traduce en una llamada a **Thread.sleep( )**. El método **timedJoin( )** es una versión especializada de **Thread.join( )** en donde *thrd* se detiene por el periodo de tiempo definido en *delay*, el cual se describe en términos de la unidad de tiempo que invoca. El método **timedWait( )** es una versión especializada de **Object.wait( )** en la cual *obj* es esperado por el periodo de tiempo especificado en *delay*, el cual está descrito en términos de la unidad de tiempo que invoca.

Las colecciones concurrentes

Como se explicó, el API de concurrencia define varias colecciones que han sido creadas para operaciones concurrentes. Ellas son:

ArrayBlockingQueue
ConcurrentHashMap
ConcurrentLinkedQueue
ConcurrentSkipListMap (Agregado por Java SE 6)
ConcurrentSkipListSet (Agregado por Java SE 6)
CopyOnWriteArrayList
CopyOnWriteArraySet
DelayQueue
LinkedBlockingDeque (Agregado por Java SE 6)
LinkedBlockingQueue
PriorityBlockingQueue
SynchronousQueue

Estas clases ofrecen alternativas concurrentes a las clases equivalentes definidas por la Estructura de Colecciones. Estas colecciones funcionan muy parecido a las otras colecciones con excepción de que proveen soporte para la concurrencia. Los programadores que estén familiarizados con la Estructura de Colecciones no tendrán problemas al utilizar estas colecciones concurrentes.

Candados

El paquete **java.util.current.locks** provee soporte para candados, los cuales son objetos que ofrecen una alternativa al uso de la palabra reservada **synchronized** para el control de acceso a un recurso compartido. En general, ésta es la forma cómo un candado funciona. Antes de acceder a un recurso compartido, el candado que protege al recurso es adquiere. Cuando el acceso al recurso termina, el candado se libera. Si un segundo hilo intenta adquirir el candado cuando está en uso por otro hilo, el segundo hilo se suspenderá hasta que el candado se libere. De esta forma, se previene un acceso conflictivo al recurso compartido.

Los candados son particularmente útiles cuando múltiples hilos necesitan tener acceso a un valor compartido. Por ejemplo, una aplicación de inventario podría tener un hilo que primero confirmara que un elemento está en el inventario y luego decrementara el número de elementos conforme ocurrieran las ventas. Si dos o más de estos hilos se están ejecutando, sin ninguna forma de sincronización, sería posible que un hilo estuviera en medio de una transacción cuando un segundo hilo comenzara su ejecución. El resultado podría ser que ambos hilos asumieran que existe producto en el inventario, incluso si no hay existencias suficientes para satisfacer una venta. En este tipo de situaciones un candado ofrece un medio conveniente para manejar la sincronización necesaria.

Todos los candados implementan de la interfaz **Lock**. Los métodos definidos por la interfaz **Lock** se muestran en la Tabla 26-1. En general, para adquirir un candado, se llama al método **lock( )**. Si el candado no está disponible, el método **lock( )** esperará. Para liberar al candado se llama al método **unlock( )**. Para revisar si un candado está disponible, y para adquirirlo si lo está, se llama al método **tryLock( )**. Este método no espera si el candado no está disponible. En lugar de eso, devuelve verdadero si consigue el candado y falso en cualquier otro caso. El método

newConditions( ) devuelve un objeto de tipo **Condition** asociado con el candado. Utilizando un objeto **Condition** se gana el control detallado del candado a través de métodos tales como **await( )** y **signal( )**, los cuales proveen una funcionalidad similar a **Object.wait** y **Object.notify( )**.

Método	Descripción
void lock( )	Espera hasta que el candado que invoca puede ser adquirido.
void lockInterruptibly( ) throws InterruptedException	Espera hasta que el candado que invoca puede ser adquirido, a menos de que sea interrumpido.
Condition newCondition( )	Devuelve un objeto de tipo **Condition** que está asociado con el candado que invoca.
boolean tryLock( )	Intenta adquirir el candado. Este método no esperará si el candado no está disponible. En lugar de esperar, regresa **true** si el candado ha sido adquirido y **false** si el candado está siendo utilizado por otro hilo.
boolean tryLock(long *esp*, TimeUnit *tu*) throws InterruptedException	Intenta establecer el candado. Si el candado no está disponible, el método esperará por un periodo no mayor al especificado por *esp*, el cual está en *tu* unidades. Este método regresa **true** si el candado ha sido adquirido y **false** si el candado no se pudo adquirir en el periodo especificado.
void unlock( )	Libera el candado.

TABLA 26-1 Los métodos de la interfaz **Lock**

java.util.concurrent.locks proporciona una implementación de la interfaz **Lock** llamada **ReentrantLock**. **ReentrantLock** implementa un *candando reentrante* que puede ser accedido en repetidas ocasiones por el hilo que posee actualmente el candado. Por supuesto, en el caso de un hilo que reingresa en un candado, todas las llamadas al método **lock( )** deben ser compensadas por un número igual de llamadas a **unlock( )**. De otra forma, un hilo buscando adquirir el candado se suspenderá hasta que el candado no esté en uso.

El siguiente programa demuestra el uso de un candado. Crea dos hilos que acceden al recurso compartido llamado **Shared.count**. Antes de que el hilo pueda acceder a **Shared.count**, debe obtener el candado. Después de obtener el candado, **Shared.count** se incrementa y luego, antes de liberar el candado, el hilo se duerme. Esto causa que el segundo hilo intente obtener el candado.

Sin embargo, dado que el candado es poseído aún por el primer hilo, el segundo hilo debe esperar hasta que el primer hilo termine de dormir y libere el candado. La salida del programa muestra que el acceso a **Shared.count** es, de hecho, sincronizado por el candado.

```
//Un ejemplo simple de candado

import java.util.concurrent.locks.*;

class LockDemo {

  public static void main(String args[]) {
    ReentrantLock lock = new ReentrantLock();

    new LockThread(lock, "A");
    new LockThread(lock, "B");

  }
}
```

```
// Un recurso compartido.
class Shared {
  static int count = 0;
}

//Un hilo de ejecución que incrementa count.
class LockThread implements Runnable {
  String name;
  ReentrantLock lock;

  LockThread(ReentrantLock lk, String n) {
    lock = lk;
    name = n;
    new Thread(this).start();
  }

  public void run() {

    System.out.println("Comenzando" + name);

    try {
      // Primer candado en acción.
      System.out.println(name + " está esperando por el candado");
      lock.lock();
      System.out.println(name + " tiene el candado.");

      Shared.count++;
      System.out.println(name + ": " + Shared.count);

      // Ahora, se permite el cambio de contexto - si es posible
      System.out.println(name + " está durmiendo");
      Thread.sleep (1000);
    } catch (InterruptedException exc) {
      System.out.println(exc);
    } finally {
      // Desbloqueo
      System.out.println(name + " está desbloqueando count.");
      lock.unlock();
    }
  }
}
```

La salida se muestra aquí. El orden exacto en el cual los hilos se ejecutan podría variar.

```
Comenzando A
A está esperando por el candado
A tiene el candado
A: 1
A está durmiendo
Comenzando B
B está esperando por el candado
A está desbloqueando count
B tiene el candado
B: 2
B está durmiendo
B está desbloqueando count.
```

java.util.concurrent.locks también define a la interfaz **ReadWriteLock**. Esta interfaz especifica un candado reentrante que mantiene candados separados para el acceso de lectura y escritura. Esto permite que sean establecidos múltiples candados para los lectores sobre un recurso mientras este recurso no esté siendo modificado. **ReentrantReadWriteLock** proporciona una implementación de **ReadWriteLock**.

Operaciones atómicas

java.util.concurrent.atomic ofrece una alternativa para las herramientas de sincronización cuando se trata de leer o escribir el valor de algunos tipos de variables. Este paquete tiene métodos que obtienen, modifican o comparan el valor de una variable en una operación ininterrumpible (esto es, en una operación atómica). Esto significa que no se requiere el uso de candados u otros mecanismos de sincronización.

Las operaciones atómicas se logran a través del uso de clases, tales como **AtomicInteger** y **AtomicLong**, y de métodos como **get( )**, **set( )**, **compareAndSet( )**, **decrementAndGet( )**, y **getAndSet( )**, los cuales realizan las acciones de (obtener, establecer, comparar y establecer, decrementar y establecer, respectivamente) como sus nombres en inglés lo indican.

El siguiente ejemplo muestra como se puede sincronizar el acceso a un dato entero compartido utilizando la clase **AtomicInteger**:

```
//Un ejemplo simple de Atomic

import java.util.concurrent.atomic.*;

class AtomicDemo{

  public static void main(String args[]) {
    new AtomThread("A");
    new AtomThread("B");
    new AtomThread("C");
  }
}

class Shared {
  static AtomicInteger ai = new AtomicInteger(0);
}

//Un hilo de ejecución
class AtomThread implements Runnable {
  String name;

  AtomThread(String n) {
    name = n;
    new Thread(this).start();
  }

  public void run() {

    System.out.println("Comenzando " + name);

    for (int i=1; i <= 3; i++)
```

```
      System.out.println(name + "obtiene: " +
                         Shared.ai.getAndSet(i));
  }
}
```

En el programa, se crea un valor estático de tipo **AtomicInteger** llamado **ai** en la clase **Shared**. Luego, se crean tres hilos de tipo **AtomThread**. Dentro del método **run( )**, se modifica **Shared.ai** al llamar al método **getAndSet( )**. Este método devuelve el valor previo y luego modifica el valor al dado como un argumento. El uso de **AtomicInteger** evita que dos hilos escriban sobre **ai** al mismo tiempo.

En general, las operaciones atómicas ofrecen una alternativa conveniente (y posiblemente más eficiente) a los mecanismos de sincronización cuando solo una variable está involucrada.

Las utilerías de concurrencia frente a la programación tradicional de Java

Dado el poder y flexibilidad que tienen las nuevas utilerías para manejo de concurrencia, es natural el hacer la siguiente pregunta: ¿Estas nuevas utilerías reemplazarán las metodologías tradicionales de Java para el manejo de multihilos y sincronización? La respuesta es un rotundo ¡no!. El soporte original para trabajo con múltiples hilos y las características predefinidas de sincronización son todavía el mecanismo que debe ser utilizado por los programas en Java, applets, y servlets. Por ejemplo, la palabra reservada **synchronized**, y los métodos **wait( )** y **notify( )** ofrecen soluciones elegantes para un gran número de problemas. Sin embargo, cuando se requiere de un mayor control, las utilerías para manejo de concurrencia están disponibles para facilitar el manejo de la situación.

27 CAPÍTULO

NES, expresiones regulares y otros paquetes

La versión original de Java incluía un conjunto de ocho paquetes que conformaban el *núcleo del API* de Java. Cada versión posterior añadió nuevos paquetes al API. Hoy en día, el API de Java contiene un gran número de paquetes. Muchos de los paquetes nuevos dan soporte a áreas especializadas que están más allá del alcance de este libro. Sin embargo, cinco de esos paquetes ameritan ser revisados aquí: **java.nio**, **java.util.regex**, **java.lang.reflect**, **java.rmi** y **java.text**. Estos paquetes dan soporte a E/S basada en NES, procesamiento de expresiones regulares, reflexión, invocación remota de métodos (RMI por sus siglas en inglés), y dar formato a texto, respectivamente.

El API de NES ofrece una forma diferente de ver y gestionar cierto tipo de operaciones de E/S. El paquete de *expresiones regulares* nos permite realizar operaciones sofisticadas de reconocimiento de patrones. Este capítulo proporciona una descripción profunda de los paquetes **java.nio**, **java.util.regex**, así como diversos ejemplos completos. La *reflexión* es la habilidad del software de analizarse a sí mismo. Es parte esencial de la tecnología Java Beans que será descrita en el Capítulo 28. La *invocación remota de métodos* (*RMI*) nos permite construir aplicaciones en Java que trabajan de manera distribuida en diversas computadoras. Este capítulo proporciona un ejemplo simple de tipo cliente/servidor que utiliza RMI. Las capacidades de *formato de texto* proporcionadas por el paquete **java.text** tienen muchos usos. En este capítulo se examina únicamente el formato de cadenas que representan fechas y horas.

El núcleo de los paquetes de Java

La Tabla 27-1 enlista los paquetes que conforman el núcleo del API de Java.

Paquete	Funcionalidad
java.applet	Soporte para la construcción de applets.
java.awt	Proporciona las capacidades necesarias para la creación de interfaces gráficas.
java.awt.color	Soporte para manejo de colores.
java.awt.datatransfer	Transferencia de datos al portapapeles del sistema.
java.awt.dnd	Soporte para las operaciones de “drag & drop”.
java.awt.event	Gestiona eventos.

TABLA 27-1 Los paquetes principales en el API de Java

Paquete	Funcionalidad
java.awt.font	Representación de varios tipos de tipografías.
java.awt.geom	Permite trabajar con figuras geométricas.
java.awt.im	Permite la escritura de caracteres japoneses, chinos y coreanos en los componentes de edición de texto.
java.awt.im.spi	Soporte para dispositivos de entrada alternativos.
java.awt.image	Procesamiento de imágenes.
java.awt.image.renderable	Soporte para el dibujo de imágenes.
java.awt.print	Soporte general para impresión.
java.beans	Permite construir componentes de software.
java.beans.beancontext	Proporciona un ambiente de ejecución para Java Beans.
java.io	Entrada y salida de datos.
java.lang	Provee la funcionalidad fundamental.
java.lang.annotation	Soporte para las anotaciones (metadatos).
java.lang.instrument	Soporte para la instrumentación.
java.lang.management	Soporte para la administración del ambiente de ejecución.
java.lang.ref	Permite interactuar con el recolector de basura de Java.
java.lang.reflect	Analiza código en tiempo de ejecución.
java.math	Gestiona números enteros y decimales grandes.
java.net	Soporte para el trabajo en red.
java.nio	Paquete principal para las clases NES.
java.nio.channels	Encapsula canales, los cuales son utilizados por el sistema NES.
java.nio.channels.spi	Soporte para canales.
java.nio.charset	Encapsula conjuntos de caracteres.
java.nio.charset.spi	Soporte para conjuntos de caracteres.
java.rmi	Provee invocación remota de métodos.
java.rmi.activation	Activa objetos persistentes.
java.rmi.dgc	Administra la recolección de basura distribuida.
java.rmi.registry	Relaciona nombres con referencias a objetos remotos.
java.rmi.server	Soporte para la invocación remota de métodos.
java.security	Gestiona certificados, llaves, compendios, firmas y otras funciones de seguridad.
java.security.acl	Administra listas de control de acceso.
java.security.cert	Analiza y administra certificados.
java.security.interfaces	Define interfaces para las llaves DSA (Digital Signature Algorithm).
java.security.spec	Especifica parámetros para llaves y algoritmos.
java.sql	Comunicación con una base de datos utilizando SQL (Structured Query Language).

TABLA 27-1 Los paquetes principales en el API de Java (*continuación*)

Paquete	Funcionalidad
java.text	Formateo, búsqueda y manipulación de texto.
java.text.spi	Soporte para formateo de texto (añadido por Java SE 6).
java.util	Contiene utilerías comunes.
java.util.concurrent	Soporte para utilerías concurrentes.
java.util.concurrent.atomic	Soporte para operaciones atómicas (esto es, indivisibles) sobre variables sin el uso de candados.
java.util.concurrent.locks	Soporte para sincronización con candados.
java.util.jar	Crear y leer archivos jar.
java.util.logging	Soporte para la creación de bitácoras con información relacionada a la ejecución del programa.
java.util.prefs	Encapsula información relacionada con las preferencias del usuario.
java.util.regex	Soporte para el procesamiento de expresiones regulares.
java.util.spi	Soporte para las clases de utilería (añadido por Java SE 6).
java.util.zip	Lee y escribe archivos zip compactados o descompactados.

TABLA 27-1 Los paquetes principales en el API de Java (*continuación*)

NES

Una adición relativamente nueva en Java es el paquete *NES* (*Nueva E/S*); éste es un paquete muy interesante que brinda soporte a una estrategia basada en canales para operaciones de E/S. Las clases NES están contenidas en los cinco paquetes mostrados a continuación:

Paquete	Propósito
java.nio	Paquete principal para el sistema NES. Encapsula varios tipos de bufer que contienen datos operados sobre el sistema NES.
java.nio.channels	Soporte para los canales, los cuales son esencialmente conexiones abiertas de E/S.
java.nio.channels.spi	Soporte para canales.
java.nio.charset	Encapsula conjuntos de caracteres. Adicionalmente soporta los codificadores y decodificadores para conversión de caracteres a bytes y de bytes a caracteres, respectivamente.
java.nio.charset.spi	Soporte para conjuntos de caracteres.

Antes de iniciar, es importante enfatizar que el subsistema NES no pretende reemplazar las clases de E/S proporcionadas por **java.io** las cuales son examinadas en el Capítulo 19. Por el contrario, las clases NES complementan el sistema estándar de E/S, dando una metodología alterna, la cual puede ser benéfica en algunas circunstancias.

Fundamentos de NES

El sistema NES está construido sobre dos elementos: bufer y canales. Un *bufer* contiene datos. Un *canal* representa una conexión abierta hacia un dispositivo de E/S, como un archivo o un socket. En

general, para utilizar el sistema NES se requiere un canal a un dispositivo de E/S y un bufer para almacenar los datos. Luego, se realizan operaciones sobre el bufer para introducir o retirar datos conforme se requiere. Las siguientes secciones examinan con más detalle los bufer y canales.

Bufer

Los bufer están definidos en el paquete **java.nio**. Todos los bufer son subclases de la clase **Buffer**, la cual define la funcionalidad base común a todos los bufer: posición actual, límite y capacidad. La *posición actual* es el índice en el bufer donde se llevará a cabo la siguiente operación de lectura o escritura. La posición actual es incrementada por la mayoría de las operaciones de lectura y escritura. El *límite* es el índice que indica el final del bufer. La *capacidad* es el número de elementos que el bufer pude almacenar. La clase **Buffer** soporta marcado y reinicialización. La clase **Buffer** define diversos métodos, los cuales son mostrados en la Tabla 27-2.

Método	Descripción
abstract Object array( )	Si el bufer que invoca está respaldado por un arreglo, se devuelve una referencia al arreglo. En caso contrario, se genera una excepción de tipo **UnsupportedOperationException**. Si el arreglo es de sólo lectura, se genera una excepción de tipo **ReadOnlyBufferException** (añadido por Java SE 6).
abstract int arrayOffset( )	Si el bufer que invoca está respaldado por un arreglo, se devuelve el índice del primer elemento. En caso contrario, se genera una excepción de tipo **UnsupportedOperationException**. Si el arreglo es de sólo lectura, se genera una excepción de tipo **ReadOnlyBufferException** (añadido por Java SE 6).
final int capacity( )	Devuelve el número de elementos que el bufer que invoca es capaz de almacenar.
final Buffer clear( )	Limpia el bufer que invoca y devuelve una referencia al bufer.
final Buffer flip( )	Establece el límite del bufer que invoca a la posición actual y reinicia la posición actual a cero. Devuelve una referencia al bufer.
abstract boolean hasArray( )	Devuelve **true** si el bufer que invoca está respaldado en un arreglo de lectura/escritura y **false** en caso contrario (añadido por Java SE 6).
final boolean hasRemaining( )	Devuelve **true** si aún quedan elementos en el bufer que invoca. Devuelve **false** en caso contrario.
abstract boolean isDirect( )	Devuelve **true** si el bufer que invoca es directo, lo cual significa que puede ser operado directamente sin necesidad de una copia. Devuelve **false** en caso contrario (añadido por Java SE 6).
abstract boolean isReadOnly( )	Devuelve **true** si el bufer que invoca es de sólo lectura. Devuelve **false** en caso contrario.
final int limit( )	Devuelve el límite del bufer que invoca.
final Buffer limit(int *n*)	Establece el límite del bufer que invoca al valor n. Devuelve una referencia al bufer.
final Buffer mark( )	Establece la marca y devuelve una referencia al bufer que invoca.
final int position( )	Devuelve la posición actual.
final Buffer position(int *n*)	Establece la posición actual del bufer que invoca al valor *n*. Devuelve una referencia al bufer que invoca.

final Buffer reset( )	Reinicia la posición actual del bufer que invoca a una marca previamente establecida. Devuelve una referencia al bufer.
final Buffer rewind( )	Establece la posición del bufer que invoca a cero. Devuelve una referencia al bufer.

TABLA 27-2 Los métodos de la clase **Buffer**

De la clase **Buffer** se derivan las siguientes clases específicas, las cuales almacenan el tipo de dato que su nombre indica:

ByteBuffer	CharBuffer	DoubleBuffer	FloatBuffer
IntBuffer	LongBuffer	MappedByteBuffer	ShortBuffer

La clase **MappedByteBuffer** es una subclase de **ByteBuffer** que se utiliza para mapear un archivo a un bufer.

Todos los bufer soportan varios métodos **get( )** y **put( )**, los cuales nos permiten obtener datos del buffer y colocar datos en el buffer. Por ejemplo, la Tabla 27-3 muestra los métodos **get( )** y **put( )** definidos por la clase **ByteBuffer**. Las otras subclases de Buffer tienen métodos similares. Todas las subclases de Buffer tienen adicionalmente métodos que ejecutan operaciones particulares sobre el bufer. Por ejemplo, el método **allocate( )** reserva espacio de manera manual para el bufer, el método **wrap( )** coloca un arreglo dentro de un bufer y el método **slice( )** crea una subsecuencia de un bufer.

Método	Descripción
abstract byte get( )	Devuelve el byte en la posición actual.
ByteBuffer get(byte *vals*[ ])	Copia al bufer que invoca en el arreglo referenciado por *vals*. Devuelve una referencia al bufer.
ByteBuffer get (byte *vals*[ ], int *inicio*, int *num*)	Copia tantos elementos como se especifique en la variable *num* del bufer que invoca en el arreglo especificado por vals, comenzando en el índice especificado por inicio. Devuelve una referencia al bufer. Si no está disponible la cantidad de elementos *num* en el bufer, se genera una excepción de tipo **BufferUnderFlowException**.
abstract byte get(int *idx*)	Devuelve el byte en la posición especificada por *idx* en el bufer que invoca.
abstract ByteBuffer put(byte *b*)	Copia el valor *b* en el bufer que invoca en la posición actual. Devuelve una referencia al bufer.
final ByteBuffer put(byte *vals*[ ])	Copia todos los elementos de *vals* en el bufer que invoca, comenzando en la posición actual. Devuelve una referencia al bufer.
ByteBuffer put(byte *vals*[ ], int *inicio*, int *num*)	Copia la cantidad de elementos especificada en la variable *num* del arreglo *vals*, comenzando en la posición especificada por inicio, al bufer que invoca. Devuelve una referencia al bufer. Si el bufer no puede almacenar todos los elementos se genera una excepción de tipo **BufferOverflowException**.
ByteBuffer put(ByteBuffer *bb*)	Copia los elementos de *bb* al bufer que invoca, comenzando en la posición actual. Si el bufer no puede almacenar todos los elementos se genera una excepción de tipo **BufferOverflowException**. Devuelve una referencia al buffer.

abstract ByteBuffer put (int *idx*, byte *b*)	Copia *b* en el bufer que invoca en la posición especificada por *idx*. Devuelve una referencia al bufer.

TABLA 27-3 Los métodos **get( )** y **put( )** definidos en la clase **ByteBuffer**

Canales

Los canales están definidos en el paquete **java.io.channels**. Un canal representa a una conexión abierta a una fuente o destino de E/S. Se obtiene un canal llamando al método **getChannel( )** sobre un objeto que soporte la aplicación de canales. Por ejemplo, el método **getChannel( )** está soportado por las siguientes clases de E/S.

DatagramSocket	FileInputStream	FileOutputStream
RandomAccessFile	ServerSocket	Socket

Así, para obtener un canal, primero se debe contar con un objeto de una de estas clases y luego llamar al método **getChannel( )** sobre dicho objeto.

El tipo específico del canal devuelto depende del tipo de objeto sobre el que se llama al método **getChannel( )**. Por ejemplo, cuando se llama a **getChannel( )** desde un objeto de tipo **FileInputStream**, **FileOutputStream** o **RandomAccessFile**, el método **getChannel( )** devuelve canales de tipo **FileChannel**. Cuando **getChannel( )** es llamado desde un objeto **Socket**, **getChannel( )** devuelve un **SocketChannel**.

Los canales como **FileChannel** y **SocketChannel** soportan varios métodos **read( )** y **write( )** que les permiten realizar operaciones de E/S a través del canal. Por ejemplo, a continuación se enlistan algunos métodos **read( )** y **write( )** definidos por **FileChannel**. Todos pueden generar excepciones de tipo **IOException**.

Método	Descripción
abstract int read(ByteBuffer *bb*)	Lee bytes del canal que invoca y los coloca en *bb* hasta que el bufer se llena o bien no existen más bytes por leer. Devuelve el número de bytes leídos.
abstract int read(ByteBuffer *bb*, long *inicio*)	Comenzando en la posición del archivo especificada por el argumento *inicio*, lee bytes del canal que invoca en *bb* hasta que el bufer se llena o bien hasta que no existen más bytes por leer. La posición actual no cambia. Devuelve el número de bytes leídos o -1 si inicio es un valor mayor al último índice del archivo.
abstract int write(ByteBuffer *bb*)	Escribe el contenido del argumento *bb* en el canal que invoca, comenzando en la posición actual. Devuelve el número de bytes escritos.
abstract int write(ByteBuffer *bb*, long *inicio*)	Comenzando en la posición del archivo especificada por el argumento *inicio*, escribe el contenido de *bb* en el canal que invoca. La posición actual no cambia. Devuelve el número de bytes escritos.

Todos los canales soportan métodos adicionales que nos dan acceso y control sobre el canal. Por ejemplo, la clase **FileChannel** cuenta con métodos para obtener y modificar la posición actual, transferir información entre canales de archivo, obtener el tamaño actual del canal, colocar un candado al canal, etc. La clase **FileChannel** además cuenta con el método **map( )**, el cual nos permite mapear un archivo a un bufer.

Conjuntos de caracteres y selectores

Otras dos entidades utilizadas por NES son los conjuntos de caracteres y los selectores. Un *conjunto de caracteres* define la forma en que los bytes son convertidos a caracteres. Se puede codificar una secuencia de caracteres como bytes utilizando un *codificador*. Luego se puede convertir una secuencia de bytes en caracteres utilizando un *decodificador*. Los conjuntos de caracteres, codificadores y decodificadores son soportados por las clases definidas en el paquete **java.nio.charset**.

Debido a que Java proporciona codificadores y decodificadores por omisión, pocas veces necesitaremos trabajar de manera explícita con conjuntos de caracteres.

Los *selectores* soportan E/S basada en llaves, sin bloqueo y multiplexada. En otras palabras, los selectores nos permiten ejecutar operaciones de E/S a través de múltiples canales. Los selectores son soportados por las clases definidas en el paquete **java.nio.channels**.
Los selectores se aplican comúnmente a canales ligados con sockets.

No utilizaremos conjuntos de caracteres ni selectores en este capítulo, pero el lector podría encontrar muy útil investigar cómo utilizarlos en sus aplicaciones.

Uso del NES

Debido a que el dispositivo de E/S más común es el disco, el resto de esta sección examina cómo acceder archivos en disco utilizando NES. Además debido a que todas las operaciones de canales conectados con archivos se basan en bytes, el tipo de bufer que utilizaremos es **ByteBuffer**.

Leyendo un archivo

Existen muchas formas de leer datos desde un archivo utilizando NES. Revisaremos dos de ellas. La primera consiste en leer el archivo manualmente reservando memoria para un bufer y luego ejecutando explícitamente las operaciones de lectura. La segunda forma utiliza un archivo de proyección, lo cual automatiza el proceso.

Para leer un archivo utilizando un canal y reservando memoria para el bufer de manera manual, se sigue el siguiente procedimiento. Primero, se abre el archivo para lectura utilizando un objeto **FileInputStream**. Luego, se obtiene un canal a este archivo llamando al método **getChannel( )**, el cual tiene la siguiente firma:

```
FileChannel getChannel( )
```

Este método devuelve un objeto **FileChannel**, el cual encapsula al canal utilizado para trabajar sobre el archivo. Una vez que se abre un canal, se obtiene el tamaño del archivo llamando al método **size( )**, mostrado a continuación:

```
long size( ) throws IOException
```

Este método devuelve el tamaño actual, en bytes, del canal, el cual refleja el tamaño del archivo subyacente. Luego, el método **allocate( )** se invoca para reservar el espacio suficiente para un bufer que pueda almacenar el contenido del archivo. Debido a que los canales de archivo operan sobre bufer de bytes se utiliza el método **allocate( )** definido en la clase **ByteBuffer**. El método tiene la siguiente firma:

```
static ByteBuffer allocate(int capacidad)
```

Donde, *capacidad* especifica la capacidad del bufer. El método devuelve una referencia al bufer. Después de crear al buffer, se llama al método **read( )** del objeto que representa al canal y se

le pasa una referencia al bufer. El siguiente programa muestra cómo leer un archivo de texto llamado **prueba.txt** a través de un canal utilizando operaciones explícitas de lectura:

```
// Utilizando NES para leer un archivo de texto
import java.io.*;
import java.nio.*;
import java.nio.channels.*;

public class ExplicitChannelRead {
  public static void main(String args[]) {
    FileInputStream fIn;
    FileChannel fChan;
    long fSize;
    ByteBuffer mBuf;

    try {
      // Primero, se abre el archivo para entrada
      fIn = new FileInputStream("prueba.txt");

      // Luego, se obtiene un canal ligado al archivo
      fChan = fIn.getChannel();

      // Ahora se obtiene el tamaño del archivo
      fSize = fChan.size();

      // Se reserva la memoria necesaria para el bufer
      mBuf = ByteBuffer.allocate((int)fSize);

      // Lee el archivo en el bufer
      fChan.read(mBuf);

      // Se coloca al inicio del bufer para leerlo
      mBuf.rewind();

      // Lee bytes del bufer
      for (int i=0; i < fSize; i++)
        System.out.println((char)mBuf.get());

      System.out.println();

      fChan.close(); // cierra el canal
      fIn.close(); //cierra el archivo
    } catch (IOException exc) {
      System.out.println(exc);
      System.exit(1);
    }
  }
}
```

El programa funciona de la siguiente manera. Primero, un archivo se abre utilizando el constructor de **FileInputStream**, y una referencia a dicho objeto es almacenada en **fIn**. Luego, un canal conectado al archivo se obtiene llamando al método **getChannel( )** sobre el objeto **fIn**, y el tamaño del archivo es obtenido llamando al método **size( )**. Después el programa llama al método **allocate( )** de la clase **ByteBuffer** para reservar memoria para un bufer que pueda almacenar el contenido del archivo. Se utiliza un bufer de bytes debido a que el objeto **FileChannel** trabaja con bytes. Una referencia a este bufer se almacena en **mBuf**. Una llamada al método **read( )** lee el contenido del archivo y lo almacena en **mBuf**. Luego el bufer es rebobinado llamando al método **rewind( )**. Esta llamada es necesaria debido a que la posición

actual de lectura está al final del bufer una vez que finalizó la ejecución del método **read( )**. Debemos colocar la posición actual al inicio del bufer para poder realizar lecturas utilizando el método **get( )**. Debido a que **mBuf** es un bufer de bytes, los valores devueltos por el método **get( )** son bytes. Los valores son convertidos a caracteres para poder mostrar el archivo como texto. De manera alternativa, es posible crear un bufer que codifique los bytes en caracteres, y luego leer caracteres de dicho bufer. El programa finaliza cerrando el canal y el archivo.

Una segunda forma, y quizá más sencilla, de leer un archivo es mapearlo en un bufer. La ventaja de hacer esto es que el bufer contiene de manera automática el contenido del archivo. No es necesario realizar de manera explícita operaciones de lectura. Para mapear y leer el contenido de un archivo se sigue el siguiente procedimiento. Primero, se abre el archivo utilizando **FileInputStream**. Luego, se obtiene un canal a dicho archivo llamando al método **getChannel** sobre el objeto. Luego se proyecta el canal a un bufer llamando al método **map( )** del objeto de tipo **FileChannel**. El método **map( )** tiene la siguiente firma:

MappedByteBuffer map(FileChannel.MapMode *m*, long *pos*, long *tam*) throws IOException

El método **map( )** hace que el contenido del archivo sea proyectado en un bufer de memoria. El valor del argumento *m* determina qué tipo de operaciones estarán permitidas; los valores válidos para m son:

MapMode.READ_ONLY	MapMode.READ_WRITE	MapMode.PRIVATE

Para leer un archivo, se utiliza **MapMode.READ_ONLY**. Para leer y escribir en el archivo se utiliza **MapMode.READ_WRITE**. **MapMode.PRIVATE** genera una copia privada del archivo de manera que los cambios realizados al bufer no afecten al archivo en disco. La posición desde la cual el archivo se comienza a proyectar en memoria se especifica con el argumento *pos*, y el número de bytes que se desean proyectar se especifica en el argumento *tam*. Una referencia al bufer es devuelta en la forma de un objeto de tipo **MappedByteBuffer**, la cual es una subclase de la clase **ByteBuffer**. Una vez que el archivo ha sido proyectado en un bufer, es posible leer el contenido del archivo desde el bufer.

El siguiente programa reconstruye el ejemplo anterior, ahora utilizando el mapeo del archivo en un bufer.

```
// Utilizando mapeo de archivos para leer un archivo de texto
import java.io.*;
import java.nio.*;
import java.nio.channels.*;

public class MappedChannelRead {
  public static void main(String args[]) {
    FileInputStream fIn;
    FileChannel fChan;
    long fSize;
    MappedByteBuffer mBuf;

    try {
      // Primero, se abre el archivo para entrada
      fIn = new FileInputStream("prueba.txt");

      // Luego, se obtiene un canal ligado al archivo
      fChan = fIn.getChannel();
```

```
      // Ahora se obtiene el tamaño del archivo
      fSize = fChan.size();

      // Ahora se proyecta el archivo en un bufer
      mBuf = fChan.map(FileChannel.MapMode.READ_ONLY, 0, fSize);

      // Lee bytes del bufer
      for (int i=0; i < fSize; i++)
        System.out.println((char)mBuf.get());

      fChan.close(); // cierra el canal
      fIn.close(); // cierra el archivo
    } catch IOException exc) {
      System.out.println(exc);
      System.exit(1);
    }
  }
}
```

Como en el ejemplo anterior, el archivo se abre utilizando un constructor de la clase **FileInputStream**, y una referencia del objeto creado se asigna a **fIn**. Se obtiene un canal conectado al archivo llamando al método **getChannel( )** desde el objeto **fIn**, y se obtiene el tamaño del archivo. Luego, el archivo completo es proyectado en la memoria llamando al método **map( ),** y una referencia al bufer se almacena en **mBuf**. Los bytes en **mBuf** se leen llamando al método **get( )**.

Escribiendo en un archivo

Existen muchas formas de escribir en un archivo utilizando canales. Nuevamente vamos a revisar dos de esas formas. La primera, se puede escribir información en un archivo utilizando canales, con operaciones explícitas de escritura. Segundo, si el archivo está abierto para lectura y escritura, se puede proyectar el archivo a un bufer y luego escribir en el bufer. Los cambios en el bufer se verán automáticamente reflejados en el archivo. Ambas formas se describen a continuación.

Para escribir en un archivo utilizando un canal y llamadas explícitas al método **write( )**, se siguen los siguientes pasos. Primero, se abre el archivo para escritura. Luego, se reserva memoria para un bufer de bytes, se colocan los datos que deseamos escribir en el bufer y luego se llama al método **write( )** del canal. El siguiente programa ejemplifica este proceso. El programa escribe el alfabeto en un archivo llamado **prueba.txt**.

```
// Utilizando NES escribir en un archivo de texto
import java.io.*;
import java.nio.*;
import java.nio.channels.*;

public class ExplicitChannelWrite {
  public static void main(String args[]) {
    FileOutputStream fOut;
    FileChannel fChan;
    ByteBuffer mBuf;

    try {
      fOut = new FileOutputStream("prueba.txt");

      // Se obtiene un canal ligado al archivo
      fChan = fOut.getChannel();
```

```
      // Se reserva la memoria necesaria para el bufer
      mBuf = ByteBuffer.allocateDirect(26);

      // Escribir algunos bytes en el bufer
      for (int i=0; i < 26; i++)
        mBuf.put((byte)('A' + i);

      // Rebobinar el archivo para almacenarlo
      mBuf.rewind();

      // Escribir el bufer en el archivo
      fChan.write(mBuf);

      // Cerrar el canal y el archivo
      fChan.close();
      fOut.close();
    } catch IOException exc) {
      System.out.println(exc);
      System.exit(1);
    }
  }
}
```

La llamada al método **rewind( )** del objeto **mBuf** es necesaria para reiniciar la posición actual a cero después de que los datos han sido escritos en **mBuf**. Recordemos que cada llamada al método **put( )** avanza la posición actual. Por lo tanto, es necesario reiniciar la posición actual al inicio del bufer antes de llamar al método **write( )**. Si no lo hacemos, el método **write( )** pensará que no hay datos en el bufer.

Para escribir en un archivo utilizando el mapeo en memoria de un archivo, se siguen los siguientes pasos. Primero, se abre el archivo para lectura / escritura. Luego, se mapea el archivo en un bufer con el método **map( )**. Después, se escribe en el bufer. Debido a que el bufer es un mapeo del archivo, cualquier cambio en el bufer se refleja automáticamente en el archivo. Por ello, no es necesario utilizar explícitamente operaciones de escritura sobre el canal. A continuación se muestra el programa anterior reconstruido para utilizar mapeo de archivos en bufer. Nótese que el archivo se abrió utilizando **RandomAccessFile**. Esto es necesario para permitir operaciones de lectura y escritura sobre el archivo.

```
// Utilizando mapeo de archivos para escribir en un archivo de texto
import java.io.*;
import java.nio.*;
import java.nio.channels.*;

public class MappedChannelWrite {
  public static void main(String args[]) {
    FileInputStream fOut;
    FileChannel fChan;
    ByteBuffer mBuf;

    try {
      fOut = new FileInputStream("prueba.txt", "rw");

      // Luego, se obtiene un canal ligado al archivo
      fChan = fOut.getChannel();

      // Ahora se proyecta el archivo en un bufer
      mBuf = fChan.map(FileChannel.MapMode.READ_WRITE, 0, 26);
```

```
      // Escribir algunos bytes en el bufer
      for (int i=0; i < 26; i++)
      mBuf.put((byte)('A' + i);

      // Cerrar el canal y el archivo
      fChan.close();
      fOut.close();
    } catch IOException exc) {
      System.out.println(exc);
      System.exit(1);
    }
  }
}
```

Como puede verse, no existen operaciones explícitas de escritura al canal. Debido a que **mBuf** está mapeado al archivo, los cambios en **mBuf** se reflejan de manera automática en el archivo subyacente.

Copiando un archivo utilizando NES

NES simplifica algunos tipos de operaciones. Por ejemplo, el siguiente programa copia un archivo. Esto se hace abriendo un canal de entrada en el archivo fuente y un canal de salida sobre el archivo destino. Luego se escribe el bufer de entrada en el archivo de salida, con una sola operación. Podemos comparar esta versión del programa con aquella realizada en el Capítulo 13. Nos daremos cuenta que la parte del programa que copia los archivos es sustancialmente más pequeña.

```
// Copiando un archivo con operaciones NES
import java.io.*;
import java.nio.*;
import java.nio.channels.*;

public class NIOCopy {

  public static void main(String args[]) {
    FileInputStream fIn;
    FileOutputStream fOut;
    FileChannel fChan, fOChan;
    long fSize;
    MappedByteBuffer mBuf;

    try {
      fIn = new FileInputStream(args[0]);
      fOut = new FileOutputStream(args[1]);

      // Canales de entrada y salida
      fIChan = fIn.getChannel();
      fOChan = fOut.getChannel();

      // Ahora se obtiene el tamaño del archivo
      fSize = fIChan.size();

      // Ahora se proyecta el archivo de entrada en un bufer
      mBuf = fIChan.map(FileChannel.MapMode.READ_ONLY, 0, fSize);

      // Escribe el bufer en el archivo de salida
      fOChan.write(mBuf); // esto copia el archivo
```

```
      // Cerrar el canal y el archivo
      fIChan.close();
      fIn.close();

      fOChan.close();
      fOut.close();
    } catch IOException exc) {
      System.out.println(exc);
      System.exit(1);
    } catch ArrayIndexOutOfBoundsException exc) {
      System.out.println("Escriba: java NIOCopy
                          archivo_fuente archivo_destino");
      System.exit(1);
    }
  }
}
```

Debido a que el archivo de entrada está mapeado en **mBuf**, **mBuf** contiene el contenido completo del archivo de entrada. Por ello, la llamada al método **write( )** copia todo el contenido de mBuf al archivo destino. Esto, por supuesto, significa que el archivo destino es una copia idéntica del archivo fuente.

¿Es NES el futuro de la gestión de operaciones de E/S?

El API de NES ofrece una forma de pensar nueva y vibrante sobre algunos tipos de operaciones relativas al manejo de archivos. Debido a ello, es natural hacerse la pregunta ¿Es NES el Futuro de la Gestión de Operaciones de E/S?. Ciertamente los canales y los bufer ofrecen una forma clara de ver la E/S. Sin embargo, agregan otra capa de abstracción. Además, la estrategia tradicional basada en flujos es, hasta la fecha, ampliamente utilizada y comprendida. Como se explicó al inicio, las operaciones de E/S basadas en canales están diseñadas actualmente para complementar, y no para remplazar, al mecanismo estándar de E/S definido en el paquete java.io. Bajo este enfoque, la estrategia de canales / bufer utilizada por el API de NES es maravillosamente exitoso. Si la nueva estrategia sustituirá algún día a la estrategia tradicional, sólo el tiempo y el uso lo dirá.

Expresiones regulares

El paquete **java.util.regex** soporta el procesamiento de expresiones regulares. El término "*expresión regular*" se utiliza en este libro para indicar una cadena de caracteres que describe a una secuencia de caracteres. Esta descripción general llamada *patrón*, puede ser utilizada para encontrar coincidencias en otras secuencias de caracteres. Las expresiones regulares pueden especificar caracteres comodines, conjuntos de caracteres, y varios cuantificadores. De esta forma, es posible especificar una expresión regular que representa una forma general que puede coincidir con varias secuencias distintas de caracteres.

Existen dos clases que soportan el procesamiento de expresiones regulares: la clase **Pattern** y la clase **Matcher**. Estas clases trabajan juntas. Se utiliza la clase **Pattern** para definir una expresión regular. Se verifica la coincidencia del patrón definido con una secuencia de caracteres utilizando un objeto de la clase **Matcher**.

Pattern

La clase **Pattern** no define constructores. En lugar de ello, un objeto de tipo Pattern se crea llamando al método de fábrica **compile( )**. Una de las firmas de este método es la siguiente:

static Pattern compile(String *patron*)

Donde, *patrón* es la expresión regular que se desea utilizar. El método **compile( )** transforma la cadena almacenada en patron en un patrón que puede ser utilizado para encontrar coincidencias mediante la clase **Matcher**. El método **compile( )** devuelve un objeto de tipo **Pattern** que contiene al patrón.

Una vez que se ha creado un objeto de tipo **Pattern**, se puede utilizar para crear un objeto **Matcher**. Esto se hace llamando al método de fábrica **matcher( )** definido por la clase **Pattern**. La firma de ese método es la siguiente:

Matcher matcher(CharSequence *str*)

Donde *str* es la secuencia de caracteres que se van a comparar con el patrón. Esta secuencia se denomina la secuencia de entrada. La interfaz **CharSequence** define un conjunto de sólo lectura de caracteres. Esta interfaz es implementada por la clase **String**, entre otras. Por ende, podemos utilizar como argumento para el método **matcher( )** un objeto de tipo **String**.

Matcher

La clase **Matcher** no tiene constructores. En lugar de ello, se crean objetos de tipo **Matcher** llamando al método de fábrica **matcher( )** definido en la clase **Pattern**, como se explicó antes. Una vez que se ha creado un objeto de tipo **Matcher**, podemos utilizar sus métodos para ejecutar diversas operaciones de búsqueda.

El método más simple de búsqueda es **matches( )**, el cual simplemente determina si una secuencia de caracteres coincide con el patrón. La firma del método es la siguiente:

boolean matches( )

Este método devuelve **true** si la secuencia y el patrón coinciden, y **false** en caso contrario. Debemos entender que la secuencia completa debe coincidir con el patrón, no sólo con una subsecuencia de él.

Para determinar si una subsecuencia de la secuencia de entrada coincide con el patrón, utilizamos el método **find( )**. Con la siguiente firma:

boolean find( )

Este método devuelve **true** si existe una subsecuencia que coincida y **false** en caso contrario. Este método puede ser llamado repetidamente, permitiéndonos encontrar todas las subsecuencias. Cada llamada a **find( )** comienza donde se detuvo la anterior.

Se puede obtener una cadena que contiene la última secuencia que coincidió con el patrón llamando al método **group( )**. Una de sus formas es la siguiente:

String group( )

La cadena que coincide con el patrón es devuelta. Si no existen coincidencias, entonces se genera una excepción de tipo **IllegalStateException**.

Se puede obtener el índice de la coincidencia actual en la secuencia de entrada llamando al método **start( )**. El índice del siguiente carácter después de la coincidencia actual se obtiene llamando al método **end( )**. Estos métodos tienen las siguientes firmas:

```
int start( )
int end( )
```

Ambos generan excepciones de tipo **IllegalStateException** si no existe ninguna coincidencia.

Se pueden remplazar todas las ocurrencias de una secuencia que coincide con otra secuencia llamando al método **replaceAll( )**, el cual tiene la siguiente firma:

```
String replaceAll(String nuevo)
```

Aquí, *nuevo* especifica la nueva secuencia de caracteres que remplazará a aquella que coincidió con el patrón. La secuencia de entrada actualizada es devuelta como una cadena.

Sintaxis de expresiones regulares

Antes de mostrar en acción a las clases **Pattern** y **Matcher**, es necesario explicar cómo construir una expresión regular. Aunque ninguna regla es complicada por sí misma, existe un gran número de ellas, y un análisis completo de las mismas va más allá del alcance de este capítulo. Sin embargo, describiremos aquí algunas de las construcciones más comunes.

En general, una expresión regular se compone de caracteres normales, clases de caracteres (conjuntos de caracteres), caracteres comodines y cuantificadores. Un carácter normal coincide tal cual. Esto es, si un patrón consiste en"xy", entonces la única secuencia de entrada que coincidirá es"xy". Los caracteres como el salto de línea y el tabulador se especifican utilizando las secuencias de escape convencionales, las cuales comienzan con el carácter \. Por ejemplo, un salto de línea se especifica como \n. En el lenguaje de las expresiones regulares, un carácter normal se denomina una literal.

Una clase de caracteres es un conjunto de caracteres. Una clase de caracteres se especifica colocando los caracteres de la clase entre corchetes. Por ejemplo la clase [wxyz] coincide con w, x, y o z. Para especificar un conjunto inverso se antepone a los caracteres un ^. Por ejemplo, [^wxyz] coincide con cualquier carácter excepto con w, x, y o z. Se puede especificar un rango de caracteres utilizando un guión. Por ejemplo para especificar una clase de caracteres que coincidirá con los dígitos del 1 al 9, se usa [1-9].

El carácter comodín es el . (punto) y coincide con cualquier carácter. Esto es, un patrón que consiste de"."coincidirá con estas (y otras) secuencias:"A","a","x", y así sucesivamente.

Un cuantificador determina cuántas veces coincide una expresión. Los cuantificadores se muestran a continuación:

+	Coincide una o más veces
*	Coincide cero o más veces
?	Coincide cero o una vez

Por ejemplo, el patrón"x+"coincidirá con"x","xx", y"xxx", entre otras.

Un último punto: en general, si se especifica una expresión no válida, se genera una excepción de tipo **PatternSyntaxException**.

Ejemplos prácticos de expresiones regulares

La mejor forma de entender cómo funcionan las expresiones regulares es revisar algunos ejemplos. El primero, mostrado a continuación, busca una coincidencia con un patrón de tipo literal:

```
// Un ejemplo simple con expresiones regulares
import java.util.regex.*;

class RegExpr {
  public static void main (String args[]) {
    Pattern pat;
    Matcher mat;
    boolean f;

    pat = Pattern.compile("Java");
    mat = pat.matcher("Java");
    f = mat.matches(); // busca una coincidencia

    System.out.println("Probando si Java coincide con Java.");
    if (f) System.out.println("Coincide");
    else System.out.println("No coincide");

    System.out.println();

    System.out.println("Probando si Java coincide con Java SE 6.");
    f = mat.matcher("Java SE 6"); // crea un nuevo Matcher

    f = mat.matches(); // busca una coincidencia

    if (f) System.out.println("Coincide");
    else System.out.println("No coincide");
  }
}
```

La salida del programa se muestra a continuación:

```
Probando si Java coincide con Java.
Coincide

Probando si Java coincide con Java SE 6.
No coincide
```

Veamos detenidamente el programa. El programa comienza creando el patrón que contiene la secuencia "Java". Luego, se crea un objeto **Matcher** para el patrón que tiene la secuencia de entrada "Java". Luego, se llama al método **matches( )** para determinar si la secuencia de entrada coincide con el patrón. Debido a que el patrón y la secuencia son iguales, el método **matches** devuelve **verdadero**. Luego, se crea un nuevo objeto **Matcher** con la secuencia de entrada "Java SE 6" y se invoca al método **matches( )** nuevamente. En este caso, el patrón y la secuencia de entrada son diferentes y no se encuentra una coincidencia. Recuerde, el método **matches( )** devuelve **verdadero** sólo si la secuencia de entrada coincide exactamente con el patrón. No devolverá **verdadero** por una coincidencia parcial.

Se puede utilizar el método **find( )** para determinar si la secuencia de entrada contiene una subsecuencia que coincida con el patrón. Consideremos el siguiente programa:

```
// Un ejemplo del método find()
import java.util.regex.*;

class RegExpr2 {
  public static void main (String args[]) {
    Pattern pat = Pattern.compile("Java");
    Matcher mat = pat.matcher("Java SE 6");

    System.out.println("Buscando Java en Java SE 6.");
```

```
    if(mat.find()) System.out.println("Subsecuencia encontrada");
    else System.out.println("No coincide");
  }
}
```

La salida se muestra a continuación:

```
Buscando Java en Java SE 6.
Subsecuencia encontrada
```

En este caso, el método **find( )** encuentra la subsecuencia "Java".

El método **find( )** puede ser utilizado para buscar en la secuencia de entrada por repetidas ocurrencias del patrón debido a que cada llamada a **find( )** comienza a buscar dónde se quedó la anterior. Por ejemplo, el siguiente programa encuentra dos ocurrencias del patrón "prueba":

```
// El método find() utilizado para encontrar múltiples subsecuencias
import java.util.regex.*;

class RegExpr3 {
  public static void main (String args[]) {
    Pattern pat = Pattern.compile("prueba");
    Matcher mat = pat.matcher("prueba 1 2 3 prueba");

    while(mat.find()) {
      System.out.println("prueba fue encontrado en la posición " + mat.start());
    }
  }
}
```

La salida se muestra a continuación:

```
prueba fue encontrado en la posición 0
prueba fue encontrado en la posición 11
```

Como lo muestra la salida, dos coincidencias son localizadas. El programa utiliza al método **start( )** para obtener el índice de cada coincidencia.

Utilizando comodines y cuantificadores

Aunque los programas anteriores muestran la técnica general para utilizar las clases **Pattern** y **Matcher**, no muestran todo su potencial. El verdadero beneficio del procesamiento de expresiones regulares no se ve hasta que se utilizan comodines y cuantificadores. Para comenzar, consideremos el siguiente ejemplo que utiliza al cuantificador + para buscar coincidencias en una secuencia arbitraria de caracteres W.

```
// Uso de cuantificadores
import java.util.regex.*;

class RegExpr4 {
  public static void main (String args[]) {
    Pattern pat = Pattern.compile("W+");
    Matcher mat = pat.matcher("W WW WWW");

    while (mat.find()) {
      System.out.println("Coincidencia: " + mat.group());
  }
}
```

La salida del programa se muestra a continuación:

```
Coincidencia: W
Coincidencia: WW
Coincidencia: WWW
```

Como lo muestra la salida, el patrón "W+" coincide con cualquier secuencia de caracteres W.

El siguiente programa utiliza un comodín para crear un patrón que coincidirá con cualquier secuencia que comience con *"a"* y termine con *"e"* para hacer esto, utilice el comodín punto junto con el cuantificador +.

```
// Uso de comodines y cuantificadores
import java.util.regex.*;

class RegExpr5 {
  public static void main (String args[]) {
    Pattern pat = Pattern.compile("a.+e");
    Matcher mat = pat.matcher("abre puerta arte fin");

    while(mat.find()) {
      System.out.println("Coincidencia: " + mat.group());
  }
}
```

Posiblemente la salida del programa anterior cause cierta sorpresa, la salida es la siguiente:

```
Coincidencia: abre puerta arte
```

Sólo una coincidencia es encontrada, y es la secuencia más larga que comienza con "a" y termina con "e". Era probable esperar dos coincidencias "abre" y "arte". La razón por la cual la secuencia más larga es encontrada es que por omisión el método **find( )** busca la secuencia más larga que coincida con el patrón. Esto se denomina *"comportamiento ambicioso"*. Podemos especificar un *"comportamiento renuente"* agregando el cuantificador **?** al patrón, como se muestra en la siguiente versión del programa. Esto causa que se obtengan las coincidencias más cortas.

```
// Uso del cuantificador ?
import java.util.regex.*;

class RegExpr6 {
  public static void main (String args[]) {
    // Utilizar un comportamiento renuente
    Pattern pat = Pattern.compile("a.+?e");
    Matcher mat = pat.matcher("abre puerta arte fin");

    while(mat.find()) {
      System.out.println("Coincidencia: " + mat.group());
  }
}
```

La salida del programa se muestra a continuación

```
Coincidencia: abre
Coincidencia: arte
```

Como lo muestra la salida, el patrón "e.+?d" coincidirá con la secuencia más corta que comience con "e" y termine con "d". Por ello se encuentran dos coincidencias en la cadena anterior.

Trabajando con clases de caracteres

Algunas veces es posible que se desee hacer coincidir una secuencia que contenga uno o más caracteres, en cualquier orden, que son parte de un conjunto de caracteres. Por ejemplo, para hacer coincidir palabras completas, desearemos hacer coincidir cualquier letra del alfabeto. Una de las formas más fáciles de hacer esto, es utilizando clases de caracteres, las cuales definen conjuntos de caracteres. Recuerde que las clases de caracteres se crean colocando los caracteres deseados entre corchetes. Por ejemplo, para hacer coincidir los caracteres en minúsculas de la 'a' a la 'z', se usa [**a-z**]. El siguiente programa demuestra esta técnica:

```
// Uso de clases de caracteres
import java.util.regex.*;

class RegExpr7 {
  public static void main (String args[]) {
    // Buscar palabras en minúsculas
    Pattern pat = Pattern.compile("[a-z]+");
    Matcher mat = pat.matcher("esto es una prueba");

    while(mat.find()) {
      System.out.println("Coincidencia: " + mat.group());
  }
}
```

La salida del programa se muestra a continuación:

```
Coincidencia: esto
Coincidencia: es
Coincidencia: una
Coincidencia: prueba
```

Utilizando replaceAll( )

El método **replaceAll( )** proporcionado por la clase **Matcher** nos permite ejecutar operaciones de búsqueda y remplazo mediante expresiones regulares. Por ejemplo, el siguiente programa remplaza todas las ocurrencias de secuencias que comienzan con los caracteres "Jon" por la secuencia "Eric".

```
// Uso del método replaceAll()
import java.util.regex.*;

class RegExpr8 {
  public static void main (String args[]) {
    String str = "Jon Jonathan Frank Ken Javier";

    Pattern pat = Pattern.compile("Jon.*? ");
    Matcher mat = pat.matcher(str);

    System.out.println("Secuencia original: " + str);

    str = mat.replaceAll("Eric ");
      System.out.println("Secuencia modificada: " + str);
  }
}
```

La salida del programa se muestra a continuación:

```
Secuencia original: Jon Jonathan Frank Ken Javier
Secuencia modificada: Eric Eric Frank Ken Javier
```

Debido a que la expresión regular "Jon.*? "coincide con cualquier cadena que comience con "Jon" seguida de cero o más caracteres y terminando con un espacio, este patrón puede ser utilizado para coincidir y remplazar tanto el texto Jon como el texto Jonathan con el texto Eric. Sustituciones como ésta no son posibles sin el uso de expresiones regulares.

Utilizando split( )

Se puede separar una secuencia de entrada en partes utilizando el método **split( )** definido en la clase **Pattern**. Una de las firmas definidas para el método **splits( )** es la siguiente:

String[ ] splits(CharSequence *str*)

Este método procesa la secuencia de entrada dada en el argumento *str*, separándola en partes acorde a los delimitadores especificados por un patrón.

Por ejemplo, el siguiente programa separa una cadena en tokens utilizando como separadores al espacio en blanco, la coma, el punto y al signo de exclamación.

```
// Uso del método split()
import java.util.regex.*;

class RegExpr9 {
  public static void main(String args[]) {

    // Buscar palabras en minúsculas
    Pattern pat = Pattern.compile("[ ,.!]");

    String strs[] = pat.split("uno dos,alfa9 12!hecho.");

    for(int i=0; i<strs.length; i++)
    System.out.println("Siguiente token: " + strs[i]);

  }
}
```

La salida del programa anterior se muestra a continuación:

```
Siguiente token: uno
Siguiente token: dos
Siguiente token: alfa9
Siguiente token: 12
Siguiente token: hecho
```

Como se muestra en la salida, la secuencia de entrada es separada en tokens. Nótese que los delimitadores no se incluyen.

Dos opciones para el método matches( )

Aunque las técnicas de búsqueda utilizando expresiones regulares, descritas anteriormente, ofrecen gran flexibilidad y potencial, existen dos alternativas que pueden resultar útiles en algunas circunstancias. Si sólo necesitamos realizar la búsqueda de una coincidencia, podemos utilizar el método **matches( )** definido en la clase **Pattern**. El cual se muestra a continuación:

static boolean matches (String *ptr*, CharSequence *str*)

El método devuelve **true** si *ptr* coincide con *str*, y **false** en caso contrario. Este método automáticamente compila el *patrón* y realiza la búsqueda. Si el patrón se va a utilizar repetidas veces entonces utilizar el método **matches( )** es menos eficiente que compilar el patrón y utilizar los métodos de búsqueda definidos en la clase **Matcher**, como se describió anteriormente.

Podemos realizar una búsqueda utilizando el método **matches( )** implementado por la clase **String**. Como se muestra a continuación:

boolean matches (String *ptr*)

Si la cadena que invoca coincide con la expresión regular dada en *ptr*, entonces el método **matches( )** devuelve **true**. En caso contrario, devuelve **false**.

Explorando las expresiones regulares

La introducción al tema de expresiones regulares presentada en esta sección sólo muestra un poco del potencial de las mismas. Dado que el análisis de texto, manipulación, y separación en tokens están ampliamente presentes en la programación de sistemas, el subsistema de expresiones regulares de Java es una herramienta sin duda importante para cualquier programador. De ahí la importancia de explorar las capacidades de las expresiones regulares. Se recomienda al lector experimentar con diferentes tipos de patrones y secuencias de entrada. Una vez que quede claro cómo trabajan las expresiones regulares, seguramente las encontrará útiles en muchos proyectos de programación.

Reflexión

La reflexión es la habilidad del software de analizarse a sí mismo. Esta capacidad es proporcionada por el paquete **java.lang.reflect** y los elementos de la clase **Class**. La reflexión es una capacidad importante, especialmente cuando se utilizan componentes de tipo Java Beans. La reflexión permite analizar un componente de software y describir dinámicamente sus capacidades, en tiempo de ejecución y no en tiempo de compilación. Por ejemplo, utilizando reflexión podemos determinar cuáles métodos, constructores, y campos contiene una clase. La reflexión fue introducida en el Capítulo 12, y será examinada a más detalle a continuación.

El paquete **java.lang.reflect** incluye diversas interfaces. Una de especial interés es la interfaz **Member**, la cual define métodos que nos permiten obtener información sobre campos, constructores o métodos de una clase. Existen ocho clases en este paquete, las cuales están listadas en la Tabla 27-4.

La siguiente aplicación ejemplifica de manera simple el uso de las capacidades de reflexión en Java. El programa muestra en pantalla los constructores, campos y métodos de la clase **java.awt.Dimension**. El programa comienza utilizando el método **forName( )** de la clase **Class** para obtener un objeto de tipo **Class** equivalente al objeto **java.awt.Dimension**. Una vez que este objeto es obtenido, se utilizan los métodos **getConstructors( )**, **getFields( )**, y **getMethods( )** para analizar el objeto. Estos métodos devuelven arreglos de objetos **Constructor**, **Field**, y **Method** que proporcionan información sobre el objeto. Las clases **Constructor**, **Field** y **Method** definen diversos métodos que pueden ser utilizados para obtener información del objeto.

Clase	Función principal
AccessibleObject	Permite pasar por alto la revisión de control de acceso por omisión.
Array	Permite crear y manipular arreglos dinámicamente.
Constructor	Proporciona información sobre un constructor.
Field	Proporciona información sobre un campo.
Method	Proporciona información sobre un método.
Modifier	Proporciona información sobre los modificadores de la clase y de acceso a miembros.
Proxy	Da soporte a clases proxy dinámicas.
ReflectPermission	Permite la reflexión de miembros privados y protegidos de una clase.

TABLA 27-4 Clases definidas en el paquete **java.lang.reflect**

Se recomienda que se experimente con cada uno de ellos. Todos ellos soportan al método **toString( )**. Por lo tanto, es posible utilizar objetos de tipo **Constructor**, **Field**, **Method** como argumentos del método **println( )**, como se muestra en el siguiente programa.

```
// Ejemplo de reflexión.
import java.lang.reflect.*;
public class ReflectionDemo1 {
  public static void main(String args[]) {
    try {
      Class c = Class.forName("java.awt.Dimension");
      System.out.println("Constructores:");
      Constructor constructors[] = c.getConstructors();
      for (int i = 0; i<constructors.length; i++) {
        System.out.println(" " + constructors[i]);
      }

      System.out.println("Campos:");
      Field fields[] = c.getFields();
      for (int i = 0; i<fields.length; i++) {
        System.out.println(" " + fields[i]);
      }

      System.out.println("Métodos:");
      Method methods[] = c.getMethods();
      for (int i=0; i<methods.length; i++) {
        System.out.println(" " + methods[i]);
      }
    } catch (Exception e) {
      System.out.println(" Error: " + e);
    }
  }
}
```

La salida del programa anterior es la siguiente. (El orden exacto puede diferir del mostrado).

```
Constructores:
 public java.awt.Dimension(int, int)
 public java.awt.Dimension()
```

```
  public java.awt.Dimension(java.awt.Dimension)
 Campos:
  public int java.awt.Dimension.width
  public int java.awt.Dimension.height
 Métodos:
  public int java.awt.Dimension.hashCode()
  public boolean java.awt.Dimension.equals(java.lang.Object)
  public java.lang.String java.awt.Dimension.toString()
  public java.awt.Dimension java.awt.Dimension.getSize()
  public void java.awt.Dimension.setSize(double, double)
  public void java.awt.Dimension.setSize(java.awt.Dimension)
  public void java.awt.Dimension.setSize(int, int)
  public double java.awt.Dimension.getHeight()
  public double java.awt.Dimension.getWidth()
  public java.lang.Object java.awt.geom.Dimension2D.clone()
  public void java.awt.geom.Dimension2D.setSize (java.awt.geom.Dimension2D)
  public final native java.lang.Class java.lang.Object.getClass()
  public final native void java.lang.Object.wait(long) throws java.lang.
    InterruptedException
  public final void java.lang.Object.wait() throws java.lang.
    InterruptedException
  public final void java.lang.Object.wait(long, int) throws java.lang.
    InterruptedException
  public final native void java.lang.Object.notify()
  public final native void java.lang.Object.notifyAll()
```

El siguiente ejemplo utiliza las capacidades de reflexión de Java para obtener los métodos públicos de una clase. El programa comienza creando un objeto de la clase **A**. Luego el método **getClass( )** es utilizado para obtener un objeto de tipo **Class** equivalente a **A**. El método **getDeclaredMethods( )** devuelve un arreglo de objetos de tipo **Method** con los métodos declarados en la clase A. Los métodos heredados no se incluyen.

Cada elemento del arreglo **mtd** es analizado. El método **getModifiers( )** devuelve un valor **entero** que describe al modificador aplicado. La clase **Modifier** proporciona un conjunto de métodos, los cuales se muestran en la tabla 27-5, que pueden ser utilizados para examinar el valor entero. Por ejemplo, el método estático **isPublic( )** devuelve **true** si su argumento incluye el modificador **public**. En caso contrario devuelve **false**. En el siguiente programa, si el método es público, el método **getName( )** obtiene su nombre y después es mostrado en pantalla.

```
// Muestra los métodos públicos
import java.lang.reflect.*;
public class ReflectionDemo2 {
  public static void main(String args[]) {

    try {
      A a = new A();
      Class c = a.getClass();
      System.out.println("Métodos públicos:");
      Method methods[] = c.getDeclaredMethods();
      for (int i=0; i<methods.length; i++) {
        int modifiers = methods[i].getModifiers();
        if (Modifier.isPublic(modifiers)) {
          System.out.println(" " + methods[i].getName());
        }
      }
    }
```

```
    catch (Exception e) {
      System.out.println(" Error: " + e);
    }
  }
}
class A {
  public void a1() {
  }
  public void a2() {
  }
  protected void a3() {
  }
  private void a4() {
  }
}
```

Esta es la salida del programa anterior:

```
Métodos públicos:
a1
a2
```

Método	Descripción
static boolean isAbstract(int *val*)	Devuelve **true** si *val* tiene activo el atributo **abstract** y **false** en caso contrario.
static boolean isFinal(int *val*)	Devuelve **true** si *val* tiene activo el atributo **final** y **false** en caso contrario.
static boolean isInterface(int *val*)	Devuelve **true** si *val* tiene activo el atributo **interface** y **false** en caso contrario.
static boolean isNative(int *val*)	Devuelve **true** si *val* tiene activo el atributo **native** y **false** en caso contrario.
static boolean isPrivate(int *val*)	Devuelve **true** si *val* tiene activo el atributo **private** y **false** en caso contrario.
static boolean isProtected(int *val*)	Devuelve **true** si *val* tiene activo el atributo **protected** y **false** en caso contrario.
static boolean isPublic(int *val*)	Devuelve **true** si *val* tiene activo el atributo **public** y **false** en caso contrario.
static boolean isStatic(int *val*)	Devuelve **true** si *val* tiene activo el atributo **static** y **false** en caso contrario.
static boolean isStrict(int *val*)	Devuelve **true** si *val* tiene activo el atributo **strict** y **false** en caso contrario.
static boolean isSynchronized(int *val*)	Devuelve **true** si *val* tiene activo el atributo **synchronized** y **false** en caso contrario.
static boolean isTransient(int *val*)	Devuelve **true** si *val* tiene activo el atributo **transient** y **false** en caso contrario.
static boolean isVolatile(int *val*)	Devuelve **true** si *val* tiene activo el atributo **volatile** y **false** en caso contrario.

TABLA 27-5 Algunos métodos definidos por la clase **Modifier**

Invocación remota de métodos (RMI)

La invocación remota de métodos (RMI por sus siglas en inglés) permite a un objeto de Java que se está ejecutando en una máquina invocar a un método de otro objeto que se está ejecutando en otra máquina. Esta es una característica importante de Java, ya que nos permite construir aplicaciones distribuidas. Aunque una discusión completa de RMI está fuera del alcance de este libro, el siguiente ejemplo describe los principios básicos involucrados.

Una aplicación cliente/servidor sencilla utilizando RMI

Esta sección proporciona instrucciones paso a paso para construir una aplicación simple cliente/servidor utilizando RMI. El servidor recibe una petición del cliente, la procesa, y regresa un resultado. En este ejemplo, la petición incluye dos números. El servidor suma los números y regresa el resultado.

Paso uno: capturar y compilar el código fuente

Esta aplicación utiliza cuatro archivos de código fuente. El primer archivo, **AddServerIntf.java** define la interfaz remota del servidor. Contiene un método que acepta dos argumentos de tipo **double** y devuelve su suma. Todas las interfaces remotas deben extender de la interfaz **Remote**, la cual es parte del paquete **java.rmi**. La interfaz **Remote** no define miembros. Su propósito es simplemente indicar que una interfaz utiliza métodos remotos. Todos los métodos remotos pueden generar excepciones de tipo **RemoteException**.

```
import java.rmi.*;
public interface AddServerIntf extends Remote {
  double add(double d1, double d2) throws RemoteException;
}
```

El segundo archivo fuente, **AddServerImpl.java** implementa la interfaz remota. La implementación del método **add( )** es simple. Todos los objetos remotos deben heredar de **UnicastRemoteObject**, la cual proporciona la funcionalidad necesaria para hacer que los objetos estén disponibles por máquinas remotas.

```
import java.rmi.*;
import java.rmi.server.*;
public class AddServerImpl extends UnicastRemoteObject
  implements AddServerIntf {

  public AddServerImpl() throws RemoteException {
  }
  public double add(double d1, double d2) throws RemoteException {
    return d1 + d2;
  }
}
```

El tercer archivo de código fuente, **AddServer.java** contiene el programa principal para la máquina servidor. Su función principal es actualizar el registro RMI en dicha máquina. Esto se realiza utilizando el método **rebind( )** de la clase **Naming** (ubicada en el paquete **java.rmi**). Este método asocia un nombre con la referencia de un objeto. El primer argumento de **rebind( )** es una cadena que identifica al servidor como "AddServer". El segundo argumento es una referencia a una instancia de **AddServerImpl**.

```
import java.rmi.*;
```

```
import java.rmi.server.*;
public class AddServer {
  public static void main(String args[]) {

    try {
      AddServerImpl addServerImpl = new AddServerImpl();
      Naming.rebind("AddServer", addServerImpl);
    } catch(Exception e) {
      System.out.println("Excepción: " + e);
    }
  }
}
```

El cuarto archivo de código fuente, **AddClient.java** implementa la aplicación cliente de este sistema distribuido. El archivo **AddClient.java** requiere de tres argumentos de línea de comando. El primero es la dirección IP o nombre de la máquina servidor. El segundo y tercer argumentos son los dos números que serán sumados.

La aplicación comienza formando una cadena que sigue la sintaxis de los URL. Esta URL utiliza el protocolo RMI. La cadena incluye la dirección IP o el nombre del servidor y la cadena "AddServer". Luego, el programa invoca al método **lookup( )** de la clase **Naming**. Este método recibe un argumento, el URL del **rmi**, y devuelve una referencia a un objeto de tipo **AddServerIntf**. Todas las invocaciones a métodos remotos pueden luego ser dirigidas a este objeto.

El programa continua desplegando sus argumentos y luego invoca al método remoto **add( )**. La suma es devuelta de este método y luego se imprime.

```
import java.rmi.*;
public class AddClient {
  public static void main(String args[]) {
    try {
      String addServerURL = "rmi://" + args[0] + "/AddServer";
      AddServerIntf addServerIntf = (AddServerIntf)Naming.lookup(addServerURL);
      System.out.println("El primer número es: " + args[1]);
      double d1 = Double.valueOf(args[1]).doubleValue();
      System.out.println("El segundo número es: " + args[2]);

      double d2 = Double.valueOf(args[2]).doubleValue();
      System.out.println("La suma es: " + addServerIntf.add(d1, d2));
    } catch (Exception e) {
      System.out.println("Excepción: " + e);
    }
  }
}
```

Después de capturar el código, utilice **javac** para compilar los cuatro archivos fuente anteriores.

Paso dos: generando un Stub

Antes de poder utilizar al cliente y al servidor, debemos generar la infraestructura necesaria, esto se logra generando lo que denominaremos stub. En el contexto de RMI, un s*tub*, es un objeto de Java que reside en la máquina del cliente. Su función es presentar las mismas interfaces que el servidor remoto. Las llamadas remotas iniciadas por el cliente son realmente dirigidas al stub. El stub trabaja con las otras partes del sistema RMI para formular la petición que es enviada a la máquina remota.

Un método remoto puede aceptar argumentos de tipos simples u objetos. En el último caso, los objetos pueden tener referencias a otros objetos. Toda esta información debe de ser enviada a la máquina remota.

Esto es, un objeto que es pasado como argumento a un método remoto deberá ser serializado y enviado a la máquina remota. Como se vio en el Capítulo 19, la serialización procesa recursivamente a todos los objetos referenciados.

Si una respuesta debe ser devuelta al cliente, el proceso trabaja a la inversa. Note que la serialización y deserialización también se utilizan para devolver un cliente.

Para generar un stub, se utiliza una herramienta denominada *compilador RMI*, la cual es invocada en línea de comando como se muestra a continuación:

```
rmic AddServerImpl
```

Este comando genera el archivo **AddServerImpl_Stub.class**. Cuando se utiliza **rmic**, debemos asegurarnos que en la variable de ambiente **CLASSPATH** está incluido el directorio actual.

Paso tres: instalar los archivos en las máquinas del cliente y el servidor

Copiemos los archivos **AddClient.class**, **AddServerImpl_Stub.class** y **AddServerIntf.class** en un directorio de la máquina cliente. Copiemos los archivos **AddServerIntf.class, AddServerImpl.class**, **AddServerImpl_Stub.class** y **AddServer.class** en un directorio de la máquina servidor.

NOTA RMI cuenta con técnicas para la carga dinámica de clases, pero éstas no son utilizadas en el ejemplo. En lugar de ello, todos los archivos que son utilizados por el cliente y el servidor deben ser instalados manualmente en cada máquina.

Paso cuatro: iniciar el registro de RMI en la máquina servidor

Java SE 6 proporciona un programa llamado **rmiregistry**, el cual se ejecuta en la máquina servidor. Este programa es responsable de relacionar nombres con referencias a objetos. Primero, revise que la variable de ambiente **CLASSPATH** incluye al directorio en el cual están ubicados los archivos. Luego, inicie el registro de RMI utilizando la siguiente línea de comando:

```
start rmiregistry
```

Una vez ejecutado el comando aparecerá una nueva ventana. Dicha ventana deberá quedar abierta mientras estemos trabajando en RMI.

Paso cinco: iniciar el servidor

El programa servidor es iniciado con la siguiente línea de comando:

```
java AddServer
```

Recuerde que el código de **AddServer** crea una instancia de **AddServerImpl** y la registra con el nombre de"AddServer".

Paso seis: iniciar el cliente

La aplicación **AddClient** requiere de tres argumentos: el nombre o dirección IP del servidor y los dos números que van a ser sumados. El cliente puede ser iniciado con una de las siguientes líneas de comando:

```
java AddClient server1 8 9
java AddClient 132.254.50.1 8 9
```

En la primera línea, se escribió el nombre del servidor. En la segunda línea se utilizó su dirección IP (132.254.50.1).

Se puede experimentar con este ejemplo sin contar con un servidor remoto. Para hacerlo, simplemente instale todos los programas en la misma computadora, inicie **rmiregistry**, inicie **AddServer**, y ejecute **AddClient** utilizando la siguiente línea de comando:

```
java AddClient 127.0.0.1 8 9
```

Donde la dirección 127.0.0.1 es la dirección que representa de manera genérica a cualquier computadora. Utilizar esta dirección nos permite experimentar con el mecanismo completo de RMI sin tener que instalar un servidor en una máquina remota.

En cualquier caso, las siguientes líneas son un ejemplo de la salida producida por el programa:

```
El primer número es: 8
El segundo número es: 9
La suma es: 17.0
```

Formato de texto

El paquete **java.text** nos permite dar formato, buscar y manipular texto. En el Capítulo 32 hay un ejemplo de la clase **NumberFormat**, la cual es utilizada para dar formato a valores numéricos. En esta sección se examinan dos de las clases de este paquete: las clases que dan formato a fechas y horas.

La clase DateFormat

La clase **DateFormat** es una clase abstracta que proporciona la habilidad de dar formato y buscar fechas y horas. El método **getDateInstance( )** devuelve una instancia de la clase **DateFormat** que puede dar formato a fechas. Este método está disponible en las siguientes formas:

```
static final DateFormat getDateInstance( )
static final DateFormat getDateInstance(int estilo)
static final DateFormat getDateInstance(int estilo, Locale localidad)
```

El argumento llamado *estilo* debe tener alguno de los siguientes valores: **DEFAULT**, **SHORT**, **MEDIUM**, **LONG** o **FULL**. Éstas son constantes de tipo **int** definidas en la clase **DateFormat**. Cada una causa diferentes efectos en la forma en que la fecha es presentada. El argumento llamado *localidad* debe tener como valor una de las referencias estáticas definidas en la clase **Locale** (véase el Capítulo 18 para mayores detalles). Si estilo o localidad no son especificados, se utilizan los valores por omisión.

Uno de los métodos más utilizados de la clase DateFormat es **format( )**. Este método tiene diversas sobrecargas, una de ellas es ésta:

```
final String format(Date d)
```

El argumento es un objeto de tipo **Date** que representa a la fecha a ser desplegada. El método devuelve una cadena que contiene la información formateada.

El siguiente ejemplo muestra cómo se utiliza la clase **DateFormat**. El ejemplo comienza creando un objeto **Date**. El objeto Date encapsula la fecha y hora actual. Luego la fecha se muestra en pantalla utilizando diferentes estilos y localidades.

```
// Ejemplo de formato de fechas
import java.text.*;
import java.util.*;

public class DateFormatDemo {
  public static void main(String args[]) {
    Date date = new Date();

    DateFormat df;

    df = DateFormat.getDateInstance(DateFormat.SHORT, Locale.JAPAN);
    System.out.println("Japón: " + df.format(date));

    df = DateFormat.getDateInstance(DateFormat.MEDIUM, Locale.KOREA);
    System.out.println("Corea: " + df.format(date));

    df = DateFormat.getDateInstance(DateFormat.LONG, Locale.UK);
    System.out.println("Reino Unido: " + df.format(date));

    df = DateFormat.getDateInstance(DateFormat.FULL, Locale.US);
    System.out.println("Estados Unidos: " + df.format(date));
  }
}
```

Un ejemplo de la salida del programa anterior es la siguiente:

```
Japón: 07/07/12
Corea: 2007.7.12
Reino Unido: 12 July 2007
Estados Unidos: Wednesday, July 12, 2007
```

El método **getTimeInstance( )** devuelve una instancia a **DateFormat** que puede dar formato a la hora. El método está disponible en las siguientes versiones:

static final DateFormat getTimeInstance( )
static final DateFormat getTimeInstance(int *estilo*)
static final DateFormat getTimeInstance(int *estilo*, Locale *localidad*)

El argumento llamado *estilo* debe tener alguno de los siguientes valores: **DEFAULT**, **SHORT**, **MEDIUM**, **LONG** o **FULL**. Éstas son constantes de tipo **int** definidas en la clase **DateFormat**. Cada una causa diferentes efectos en la forma en que la fecha es presentada. El argumento llamado *localidad* debe tener como valor una de las referencias estáticas definidas en la clase **Locale**. Si *estilo* o *localidad* no son especificados, se utilizan los valores por omisión.

El siguiente ejemplo muestra cómo se utiliza la clase **DateFormat** para dar formato a horas. El ejemplo comienza creando un objeto **Date**. El objeto Date encapsula la fecha y hora actual. Luego la hora se muestra en pantalla utilizando diferentes estilos y localidades.

```
// Ejemplo de formato de hora
import java.text.*;
import java.util.*;
public class TimeFormatDemo {
```

```
  public static void main(String args[]) {
    Date date = new Date();
    DateFormat df;

    df = DateFormat.getTimeInstance(DateFormat.SHORT, Locale.JAPAN);
    System.out.println("Japón: " + df.format(date));
    df = DateFormat.getTimeInstance(DateFormat.LONG, Locale.UK);
    System.out.println("Reino Unido: " + df.format(date));

    df = DateFormat.getTimeInstance(DateFormat.FULL, Locale.CANADA);
    System.out.println("Canadá: " + df.format(date));
  }
}
```

Un ejemplo de la salida del programa anterior es la siguiente:

```
Japón: 20:25
Reino Unido: 20:25:14 CDT
Canadá: 8:25:14 o´clock PM CDT
```

La clase **DateFormat** define también al método **getDateTimeInstance( )** que puede formatear tanto fechas como horas. Se deja al lector experimentar con este método por él mismo.

La clase SimpleDateFormat

La clase **SimpleDateFormat** es una subclase concreta de la clase **DateFormat**. Esta clase nos permite definir nuestros propios patrones de formato, los cuales se utilizarán para desplegar información de fecha y hora.

Uno de sus constructores se muestra a continuación:

SimpleDateFormat(String *formato*)

El argumento *formato* describe cómo será desplegada la información. El siguiente es un ejemplo de cómo se utiliza el método anterior:

```
SimpleDateFormat sdf = SimpleDateFormat("dd MMM yyyy hh:mm:ss zzz");
```

Los símbolos utilizados en la cadena de formato determinan la información que será desplegada. La Tabla 27-6 lista los símbolos que pueden ser utilizados y da una descripción de cada uno.

En muchos casos, el número de veces que un símbolo es repetido determina cómo se presenta ese dato. La información textual se presenta en una forma abreviada si el patrón de letras se repite menos de cuatro veces. En caso contrario, se utiliza la forma no abreviada. Por ejemplo, un patrón "zzzz" despliega "Pacific Daylight Time" mientras que un patrón "zzz" despliega "PDT".

Para números, la cantidad de veces que un símbolo se repite en el patrón determina cuántos dígitos serán presentados. Por ejemplo, "hh:mm:ss" podría presentar "01:51:15", pero "h:m:s" desplegaría el mismo valor como "1:51:15".

Finalmente, "M" o "MM" despliegan el mes con uno o dos dígitos respectivamente. Sin embargo, tres o más repeticiones de M causan que el mes sea mostrado como una cadena de texto.

El siguiente programa muestra cómo se utiliza la clase SimpleDateFormat:

```
// Ejemplo de SimpleDateFormat
import java.text.*;
import java.util.*;
```

```
public class SimpleDateFormatDemo {
  public static void main(String args[]) {
    Date date = new Date();
    SimpleDateFormat sdf;
    sdf = new SimpleDateFormat("hh:mm:ss");
    System.out.println(sdf.format(date));
    sdf = new SimpleDateFormat("dd MMM yyyy hh:mm:ss zzz");
    System.out.println(sdf.format(date));
    sdf = new SimpleDateFormat("E MMM dd yyyy");
    System.out.println(sdf.format(date));
    System.out.println(sdf.format(date));
  }
}
```


La salida del programa es la siguiente

```
10:23:03
12 Jul 2007 10:25:03 CDT
Wed Jul 12 2007
```

TABLA 27-6 Símbolos para formato de cadenas en la clase **SimpleDateFormat**

Símbolo	Descripción
a	AM o PM
d	Día del mes (1-31)
h	Hora en formato AM/PM (1-12)
k	Hora del día (1-24)
m	Minutos en la hora (0-59)
s	Segundos en el minuto (0-59)
w	Semana en el año (1-52)
y	Año
z	Zona horaria
D	Día en el año (1-366)
E	Día en la semana (por ejemplo Martes)
F	Día en la semana del mes
G	Era (esto es AD o BC)
H	Hora en el día (0-23)
K	Hora en formato AM/PM (0-11)
M	Mes
S	Milisegundo en el segundo
W	Semana en el mes (1-5)
Z	Zona horaria en formato RFC822

PARTE III

Desarrollo de software utilizando Java

28 CAPÍTULO

Java Beans

Este capítulo proporciona una visión general de Java Beans. Los Beans son importantes porque permiten crear sistemas complejos a partir de componentes de software. Estos componentes pueden ser construidos por el programador o abastecidos por uno o más vendedores diferentes. Java Beans define una arquitectura que especifica cómo estos bloques pueden operar juntos.

Para comprender mejor el valor de los Beans, considere lo siguiente. Los diseñadores de hardware tienen una gran variedad de componentes que pueden ser integrados conjuntamente para construir un sistema. Resistencias, capacitores e inductores son ejemplos de bloques simples de construcción. Los circuitos integrados proveen una funcionalidad avanzada. Todas estas partes diferentes pueden ser reutilizadas. No es necesario reconstruir estas partes cada vez que se requieren en un sistema nuevo. Además, las mismas piezas pueden ser utilizadas en diferentes tipos de circuitos. Esto es posible debido a que el comportamiento de estos componentes se entiende y está documentado.

La industria del software también ha buscando los beneficios de la reusabilidad e interoperabilidad de una estrategia basada en componentes. Para obtener esos beneficios, es necesaria una arquitectura basada en componentes que permita a los programas ser ensamblados a partir de bloques de software; bloques que incluso puedan ser proporcionados por diferentes vendedores. Debe también ser posible para un diseñador elegir un componente, entender sus capacidades e incorporarlo dentro de una aplicación. Cuando una nueva versión de un componente está disponible, debería ser sencillo incorporar su funcionalidad dentro del código existente. Afortunadamente, Java Beans proporciona tal arquitectura.

¿Qué es Java Beans?

Un *Java Bean* es un componente de software que ha sido diseñado para ser reutilizable en una gran variedad de ambientes diferentes. No hay restricciones en la capacidad de un Bean. Podría ejecutar una función simple, tal como obtener un valor de un inventario, o una función compleja, tal como pronosticar el rendimiento de un portafolio de acciones. Un Bean podría ser visible para un usuario final. Un ejemplo de esto es un botón sobre una interfaz gráfica de usuario. Sin embargo, un Bean también podría ser invisible para el usuario. Un ejemplo de este tipo de bloque sería un software para decodificar un flujo de información multimedia en tiempo real. Finalmente, un Bean podría estar diseñado para trabajar autónomamente sobre la estación de trabajo de un usuario o trabajar en cooperación con un conjunto de otros componentes distribuidos. Un ejemplo de un Bean que se puede ejecutar localmente es un software para generar un diagrama de pastel sobre un conjunto de

datos. Sin embargo, un Bean que provee información en tiempo real del valor de acciones en la bolsa o venta de artículos de consumo necesitará trabajar en cooperación con otros sistemas de software para obtener esos datos.

Ventajas de los Java Beans

La siguiente lista enumera algunos de los beneficios que la tecnología de Java Beans provee para un desarrollador de componentes:

- Un Bean obtiene todos los beneficios del paradigma de Java"escríbelo una vez, córrelo en cualquier lugar".
- Las propiedades, eventos y métodos de un Bean a los que tiene acceso otra aplicación puede ser controlados.
- Se puede proveer software auxiliar para ayudar a configurar un Bean. Este software sólo es necesario para definir los parámetros en el diseño para ese componente. No es necesario que se incluya, el software auxiliar, en el ambiente de ejecución.
- Las propiedades de configuración de un Bean pueden ser almacenadas de manera persistente y recuperadas tiempo después.
- Un Bean puede recibir eventos generados por otros objetos y puede también generar eventos que sean enviados a otros objetos.

Introspección

En el núcleo de Java Beans está el concepto de *introspección*. Esto es el proceso de analizar un Bean para determinar sus capacidades. Ésta es una característica esencial del API de Java Beans, porque permite a otras aplicaciones, tal como una herramienta de diseño, obtener información acerca de un componente. Sin la introspección, la tecnología Java Beans no podría operar.

Hay dos formas mediante las cuales el desarrollador de un Bean puede indicar cual de las propiedades, eventos, y métodos del Bean deberían estar a la vista. Con el primer método, simplemente se utilizan convenciones para los nombres. Esto permite al mecanismo de introspección inferir información acerca del Bean. En la segunda forma, una clase adicional que extiende de la interfaz **BeanInfo** se define para proporcionar explícitamente información del Bean. Ambas estrategias se examinan a continuación.

Patrones de diseño para propiedades

Una *propiedad* describe a un subconjunto del estado de un Bean. Los valores asignados a las propiedades determinan el comportamiento y la apariencia del componente. Una propiedad se define mediante un método *setter*. Una propiedad se obtiene mediante un método *getter*. Hay dos tipos de propiedades: simples e indexadas.

Propiedades simples

Una propiedad simple, tiene un valor único. El cual puede ser identificado por los siguientes patrones de diseño, donde **N** es el nombre de la propiedad y **T** es el tipo:

```
public T getN( )
public void setN(T arg)
```

Una propiedad de lectura y escritura tiene ambos métodos para acceder a sus valores. Una propiedad de sólo lectura tiene solamente el método get. Una propiedad de sólo escritura tiene solamente el método set.

El siguiente código define tres propiedades de lectura y escritura con sus respectivos métodos get y set:

```
private double depth, height, width;

public double getDepth( ) {
  return depth;
}
public void setDepth(double d) {
  depth = d;
}

public double getHeight( ) {
  return height;
}
public void setHeight(double h) {
  height = h;
}

public double getWidth( ) {
  return width;
}
public void setWidth(double w) {
  width = w;
}
```


Propiedades indexadas

Una propiedad indexada consiste en valores múltiples. Puede ser identificada por los siguientes patrones de diseño, donde **N** es el nombre de la propiedad y **T** es su tipo:

public T getN(int *index*);
public void setN(int *index,* T *valor*);
public T[ ] getN( );
public void setN(T *values*[ ]);

Aquí está una propiedad indexada llamada **data** junto con sus métodos get y set:

```
private double data[ ];

public double getData(int index) {
  return data [index];
}
public void setData(int index, double value) {
  data[index] = value;
}
public double[ ] getData( ) {
  return data;
}
public void setData(double[ ] values) {
  data = new double[values.length];
  System.arraycopy(values, 0, data, 0, values.length);
}
```

Patrones de diseño para eventos

Los Beans utilizan el modelo de delegación de eventos que fue discutido anteriormente en este libro. Los Beans pueden generar eventos y enviarlos a otros objetos. Éstos pueden ser identificados por los siguientes patrones de diseño, donde **T** es el tipo del evento:

public void addTListener(TListener *eventListener*)
public void addTListener(TListener *eventListener*)
throws java.util.TooManyListenersException
public void removeTListener(TListener *eventListener*)

Estos métodos son utilizados para agregar o eliminar un listener para el evento especificado. La versión de **AddTListener( )** que no lanza una excepción puede ser utilizado para hacer *difusión múltiple* de un evento, lo cuál significa que más de un listener puede escuchar la notificación del evento. La versión que lanza **TooManyListenersException** hace *mono difusión* del evento, lo cuál significa que el número de listeners se restringe a uno. En cualquier caso, **removeTListener( )** se utiliza para eliminar a los listeners. Por ejemplo, asumiendo que existe una interfaz de eventos de tipo **TemperatureListener**, un Bean que monitorea la temperatura podría proveer los siguientes métodos:

```
public void addTemperatureListener(TemperaturaListener tl) {
. . .
}
public void removeTemperatureListener(TemperaturaListener tl) {
. . .
}
```

Métodos y patrones de diseño

Los patrones de diseño no se utilizan para nombrar a los métodos que no representan propiedades. El mecanismo de introspección encuentra todos los métodos públicos de un Bean. Los métodos protected y private no se presentan.

Uso de la interfaz BeanInfo

Como se mostró en la discusión anterior, los patrones de diseño *implícitamente* determinan qué información está disponible para el usuario de un Bean. La interfaz **BeanInfo** habilita al programador para controlar explícitamente la información disponible. La interfaz **BeanInfo** define varios métodos, incluyendo los siguientes:

PropertyDescriptor[ ]getPropertyDescriptors( )
EventSetDescriptor[ ]getEventSetDescriptors( )
MethodDescriptor[ ]getMethodDescriptors( )

Estos métodos devuelven un arreglo de objetos que proporcionan información acerca de las propiedades, eventos y métodos de un Bean. Las clases **PropertyDescriptor**, **EventSetDescriptor**, y **MethodDescriptor** se definen dentro del paquete **java.beans**, y describen el elemento indicado. Implementando estos métodos, un desarrollador puede designar exactamente lo que se presenta al usuario, dejando de lado la introspección basada en patrones de diseño.

Cuando se crea una clase que implementa la interfaz **BeanInfo**, se debe llamar a la clase *bnombre*BeanInfo, donde *bnombre* es el nombre del Bean. Por ejemplo, si el Bean es llamado **MyBean**, entonces la información de la clase debe ser llamada **MyBeanBeanInfo**.

Para simplificar el uso de **BeanInfo**, los Java Beans proporcionan la clase **SimpleBeanInfo**, la cual provee la implementación por omisión de la interfaz **BeanInfo**, incluyendo los tres métodos mostrados antes. Se puede extender esta clase y sobrescribir uno o más de los métodos para controlar explícitamente los aspectos que serán expuestos del Bean. Si no se sobrescribe un método, entonces la introspección basada en patrones de diseño será utilizada. Por ejemplo, si no se sobrescribe **getPropertyDescriptors( )**, entonces los patrones de diseño se utilizan para descubrir las propiedades del Bean. Más adelante veremos a **SimpleBeanInfo** en acción.

Propiedades limitadas y restringidas

Un Bean que tiene una propiedad *limitada* genera un evento cuando la propiedad cambia. El evento es de tipo **PropertyChangeEvent** y se envía a los objetos que previamente registraron su interés en recibir tales notificaciones. Una clase que gestiona este evento debe implementar la interfaz **PropertyChangeListener**.

Un Bean que tiene una propiedad *restringida*, genera un evento cuando se hace un intento de cambiar su valor. También genera un evento de tipo **PropertyChangeEvent**. Este evento también se envía a los objetos que previamente registraron su interés en recibir tales notificaciones. Sin embargo, estos otros objetos tienen la habilidad de vetar el cambio propuesto generando una excepción de tipo **PropertyVetoException**. Esta capacidad permite a un Bean funcionar de manera diferente de acuerdo con su ambiente de ejecución. Una clase que gestiona este tipo de eventos debe implementar la interfaz **VetoableChangeListener**.

Persistencia

La *persistencia* es la habilidad de guardar el estado actual de un Bean, incluyendo los valores de las propiedades y variables de instancia, en un almacenamiento no volátil y recuperarlos posteriormente. La capacidad de serialización de objetos proporcionada por las bibliotecas de clases de Java se utiliza para proporcionar persistencia a los Beans.

La forma más fácil de serializar un Bean es hacer que implemente la interfaz **java.io.Serializable**, la cual es simplemente una interfaz de marcado. Implementando la interfaz **java.io.Serializable** permite la serialización automática. El Bean no necesita tomar ninguna otra acción. La serialización automática también puede ser heredada. Por consiguiente, si alguna superclase de un Bean implementa la interfaz **java.io.Serializable**, entonces automáticamente se obtiene la serialización para el Bean. Existe una restricción importante: cualquier clase que implementa de **java.io.Serializable** debe suministrar un constructor sin parámetros.

Cuando se utiliza serialización automática, es posible prevenir selectivamente que un campo sea guardado utilizando la palabra clave **transient**. De esta forma, los datos miembro de un Bean especificados como **transient** no serán serializados.

Si un Bean no implementa la interfaz **java.io.Serializable**, el programador deberá proveer la serialización el mismo, por ejemplo implementado la interfaz **java.io.Externalizable**. De otra forma, los contenedores no almacenarán la configuración del componente.

Customizers

Un desarrollador de Beans puede proporcionar un *customizer* que ayude a otro desarrollador a configurar al Bean. Un customizer puede proporcionar una guía paso a paso del proceso que se

debe seguir para utilizar el componente en un contexto específico. También se puede proporcionar documentación en línea. Un desarrollador de Beans dispone de gran flexibilidad para desarrollar un customizer que puede diferenciar su producto en el mercado.

La Java Beans API

La funcionalidad de Java Beans es proporcionada por un conjunto de clases e interfaces en el paquete **java.beans**. En esta sección se da una breve descripción general de su contenido. La Tabla 28-1 enlista las interfaces de **java.beans** y proporciona una breve descripción de su funcionalidad. La Tabla 28-2 enlista las clases en el paquete **java.beans**.

Aunque está fuera del alcance de este capítulo discutir todas las clases, cuatro son de especial interés: **Introspector**, **PropertyDescriptor**, **EventSetDescriptor** y **MethodDescriptor**. Cada una se examina brevemente a continuación.

Interfaz	Descripción
AppletInitializer	Los métodos en esta interfaz son utilizados para inicializar Beans que son también applets.
BeanInfo	Esta interfaz permite a un diseñador especificar información acerca de las propiedades, eventos y métodos de un Bean.
Customizer	Esta interfaz permite a un diseñador proporcionar una interfaz gráfica de usuario a través de la cuál un Bean puede ser configurado.
DesignMode	Los métodos de esta interfaz determinan si un Bean se ejecuta en modo de diseño.
ExceptionListener	Un método de esta interfaz se invoca cuando una excepción ha ocurrido.
PropertyChangeListener	Un método de esta interfaz se invoca cuando una propiedad limitada es modificada.
PropertyEditor	Los objetos que implementan esta interfaz permiten a los diseñadores cambiar y mostrar los valores de las propiedades.
VetoableChangeListener	Un método en esta interfaz se invoca cuando una propiedad restringida es modificada.
Visibility	Los métodos en esta interfaz permiten a un Bean ejecutarse en ambientes donde una interfaz gráfica de usuario no está disponible.

TABLA 28-1 Las interfaces en el paquete **java.beans**

Clase	Descripción
BeanDescriptor	Esta clase provee información acerca de un Bean. También permite asociar un customizer con un Bean.
Beans	Esta clase se utiliza para obtener información acerca de un Bean.
DefaultPersistenceDelegate	Una subclase concreta de **PersistenceDelegate**.
Encoder	Codifica el estado de un grupo de Beans. Puede ser utilizada para escribir esta información en un flujo.
EventHandler	Soporta creación dinámica de listener de eventos.

TABLA 28-2 Las clases en el paquete **java.beans**

Clase	Descripción
EventSetDescriptor	Las instancias de esta clase describen un evento que puede ser generado por un Bean.
Expression	Encapsula una llamada a un método que devuelve un resultado.
FeatureDescriptor	Ésta es la superclase de las clases **PropertyDescriptor**, **EventSetDescriptor** y **MethodDescriptor**.
IndexedPropertyChangeEvent	Una subclase de **PropertyChangeEvent** que representa un cambio de una propiedad indexada.
IndexedPropertyDescriptor	Las instancias de esta clase describen una propiedad indexada de un Bean.
IntrospectionException	Una excepción de este tipo se genera si un problema ocurre cuando se analiza un Bean.
Introspector	Esta clase analiza un Bean y construye un objeto **BeanInfo** que describe el componente.
MethodDescriptor	Las instancias de esta clase describen un método de un Bean.
ParameterDescriptor	Las instancias de esta clase describen a los parámetros de un método.
PersistenceDelegate	Maneja la información del estado de un objeto.
PropertyChangeEvent	Este evento es generado cuando propiedades limitadas o restringidas son modificadas. Este evento se envía a los objetos que registraron interés en él y que implementan, ya sea de la interfaz **PorpertyChangeListener** o de **VetoableChangeListener**.
PropertyChangeListenerProxy	Extiende de **EventListenerProxy** e implementa **PropertyChangeListner**.
PropertyChangeSupport	Los beans que soportan propiedades limitadas pueden utilizar esta clase para notificar a objetos **PropertyChangeListener**.
PropertyDescriptor	Las instancias de esta clase describen una propiedad de un Bean.
PropertyEditorManager	Esta clase localiza un objeto **PropertyEditor** para un tipo dado.
PropertyEditorSupport	Esta clase proporciona la funcionalidad que puede ser utilizada cuando se escriben editores de propiedades.
PropertyVetoException	Una excepción de este tipo se genera si un cambio en una propiedad restringida es vetado.
SimpleBeanInfo	Esta clase provee funcionalidad que puede ser utilizada cuando se escriben clases **BeanInfo**.
Statement	Encapsula la llamada a un método.
VetoableChangeListenerProxy	Extiende de **EventListenerProxy** e implementa de **VetoChangeListener**.
VetoableChangeSupport	Los Beans que soportan propiedades restringidas pueden utilizar esta clase para notificar a objetos de tipo **VetoableChangeListener**.
XMLDecoder	Se utiliza para leer un Bean desde un documento XML.
XMLEncoder	Se utiliza para escribir un Bean en un documento XML.

TABLA 28-2 Las clases en el paquete **java.beans** (*continuación*)

Introspector

La clase **Introspector** proporciona varios métodos estáticos que soportan introspección. El de mayor interés es **getBeanInfo( )**. Este método devuelve un objeto **BeanInfo** que puede ser utilizado para obtener información acerca del Bean. El método **getBeanInfo( )** tiene varias formas, incluyendo la que se muestra a continuación:

static BeanInfo getBeanInfo(Class<?>*bean*) throws InstrospectionException

El objeto devuelto contiene la información acerca del Bean especificado por *bean*.

PropertyDescriptor

La clase **PropertyDescriptor** describe una propiedad de un Bean. Soporta varios métodos que manejan y describen propiedades. Por ejemplo, se puede determinar si una propiedad es limitada al llamar al método **isBound( )**. Para determinar si una propiedad es restringida se llama al método **isConstrained( )**. Se puede obtener el nombre de la propiedad llamando al método **getName( )**.

EventSetDescriptor

La clase **EventSetDescriptor** representa un evento de un Bean. Soporta varios métodos que obtienen los métodos que un Bean utiliza para agregar o eliminar listeners de eventos y de otras formas de manejar eventos. Por ejemplo, para obtener el método utilizado para agregar listeners, se llama al método **getAddListenerMethod( )**. Para obtener el método utilizado para eliminar listeners, se llama al método **getRemoveListenerMethod( )**. Para obtener el tipo de un listener, se llama al método **getListenerType( )**. Se puede obtener el nombre de un evento llamando al método **getName( )**.

MethodDescriptor

La clase **MethodDescriptor** representa un método en un Bean. Para obtener el nombre del método, se llama a **getName( )**. Se puede obtener información acerca del método llamando al método **getMethod( )**, mostrado a continuación:

Method getMethod( )

El método devuelve un objeto de tipo **Method** que describe al método del Bean.

Un ejemplo de programación de Java Beans

Este capítulo concluye con un ejemplo que ilustra varios aspectos de programación de Beans, incluyendo introspección y uso de la clase **BeanInfo**. También muestra el uso de las clases **Introspector**, **PropertyDescriptor** y **EventSetDescriptor**. El ejemplo utiliza tres clases. La primera que es un Bean llamado **Colores**, se muestra a continuación:

```
// Un Bean simple
import java.awt.*;
import java.awt.event.*;
import java.io.Serializable;

public class Colores extends Canvas implements Serializable {
  transient private Color color; // no persistente
  private boolean rectangular; // es persistente
```

```
  public Colores() {
    addMouseListener(new MouseAdapter() {
      public void mousePressed(MouseEvent me) {
        change();
      }
    });
    rectangular = false;
    setSize (200, 100);
    change();
  }

  public boolean getRectangular() {
    return rectangular;
  }

  public void setRectangular(boolean flag) {
    this.rectangular = flag;
    repaint() ;
  }

  public void change() {
    color = randomColor();
    repaint();
  }

  private Color randomColor() {
    int r = (int) (255*Math.random());
    int g = (int) (255*Math.random());
    int b = (int) (255*Math.random());
    return new Color(r, g, b);
  }

  public void paint(Graphics g) {
    Dimension d = getSize();
    int h = d.height;
    int w = d.width;
    g.setColor(color);
    if(rectangular) {
      g.fillRect(0, 0, w-1, h-1);
    }else {
      g.fillOval(0, 0, w-1, h-1);
    }
  }
}
```

El Bean **Colores** despliega un objeto coloreado dentro de un marco. El **color** del componente está determinado por la variable privada **color**, y su forma está determinada por la variable privada llamada **rectangular**. El constructor define una clase interior anónima que extiende de **MouseAdapter** y sobrescribe su método **mousePressed( )**. El método **change( )** se invoca en respuesta al evento de presionar el botón del ratón, este método selecciona un color aleatorio y luego pinta de nuevo el componente. Los métodos **getRectangular( )** y **setRectangular( )** proveen acceso a la propiedad del Bean. El método **change( )** llama al método **randomColor( )** para seleccionar el color y luego llama a **repaint( )** para hacer el cambio visible. Nótese que el método **paint( )** utiliza las variables rectangular y color para determinar cómo presentar el Bean.

La siguiente clase es **ColoresBeanInfo**. Es una subclase de **SimpleBeanInfo** que provee explícitamente información acerca del Bean llamado Colores. La clase **ColoresBeanInfo** sobrescribe al método **getPropertyDescriptors( )** para designar las propiedades que son presentadas al usuario en un Bean. En este caso, la única propiedad expuesta es rectangular. El método crea y devuelve un objeto **PropertyDescriptor** para la propiedad llamada rectangular. El constructor **PropertyDescriptor** que se utilizó se muestra a continuación:

PropertyDescriptor(String *property*, Class <?>*beanCls*) throws IntrospectionException

Aquí, el primer argumento es el nombre de la propiedad, y el segundo argumento es la clase del Bean.

```
// Una clase Bean de información.
import java.beans.*;
public class ColoresBeanInfo extends SimpleBeanlnfo {
  public PropertyDescriptor[] getPropertyDescriptors(){
    try {
      PropertyDescriptor rectangular = new
        PropertyDescriptor("rectangular", Colores.class);
      PropertyDescriptor pd[] = {rectangular};
      return pd;
    } catch(Exception e) {
      System.out.println("Excepción atrapada: " + e);
    }
    return null;
  }
}
```

La última clase se llamada **IntrospectorDemo**. Esta clase utiliza introspección para mostrar las propiedades y eventos que están disponibles dentro del Bean **Colores**.

```
// Muestra las propiedades y eventos.
import java.awt.*;
import java.beans.*;

public class IntrospectorDemo {
  public static void main(String args[]){
    try {
      Class c = Class.forName ("Colores") ;
      BeanInfo beanInfo = Introspector.getBeanlnfo(c);

      System.out.println("Propiedades:") ;
      PropertyDescriptor propertyDescriptor[] =
        beanlnfo.getPropertyDescriptors() ;
      for(int i = 0; i < propertyDescriptor.length; i++) {
        System.out.println("\t" + propertyDescriptor[i].getName());
      }

      System.out.println ("Eventos: ");
      EventSetDescriptor eventSetDescriptor[] =
        beanlnfo.getEventSetDescriptors() ;
      for(int i = 0; i < eventSetDescriptor.length; i++) {
        System.out.println("\t" + eventSetDescriptor[i].getName());
      }
```

```
    } catch(Exception e) {
      System.out.println("Excepción atrapada: " + e);
    }
  }
}
```

La salida de este programa es la siguiente:

```
Propiedades:
         rectangular
Eventos:
         mouseWheel
         mouse
         mouseMotion
         component
         hierarchyBounds
         focus
         hierarchy
         propertyChange
         inputMethod
         key
```

Es importante observar dos cosas en la salida. Primero, **ColoresBeanInfo** sobrescribe al método **getPropertyDescriptor( )** de tal forma que sólo la propiedad **rectangular** es devuelta por el método y por tanto, sólo la propiedad rectangular es visible. Sin embargo, debido a que el método **getEventSetDescriptors( )** no está sobrescrito en la clase **ColoresBeanInfo**, la introspección basada en patrones de diseño se aplica, y todos los eventos se despliegan, incluidos aquellos que pertenecen a la clase **Canvas** que es la clase padre de **Colores**. Recuerde que si no se sobrescribe uno de los métodos "get" definidos por **SimpleBeanInfo**, entonces, se utiliza la introspección por omisión basada en patrones de diseño. Para observar la diferencia que **ColoresBeanInfo** hace, borre su archivo clase y entonces ejecute **IntrospectorDemo** de nuevo. Esta vez se reportarán más propiedades.

29 CAPÍTULO

Introducción a Swing

En la Parte II, vimos cómo construir interfaces de usuario con las clases de AWT. Aunque AWT aún es una parte crucial de Java, su conjunto de componentes ya no es utilizado tan ampliamente para crear interfaces de usuario. Hoy en día, muchos programadores utilizan Swing para este propósito. Swing es un conjunto de clases que proporcionan componentes más poderosos y flexibles que los de AWT. Dicho de manera simple, con Swing se construye las interfaces gráficas modernas de Java.

Swing se describe en dos capítulos. Este capítulo es una introducción a Swing. Comienza con la descripción de los conceptos clave. Luego se muestra la forma general de un programa incluyendo tanto aplicaciones como Applets.Y concluye explicando como dibujar en Swing. El siguiente capítulo presenta diversos componentes de Swing utilizados comúnmente. Es importante entender que el número de clases e interfaces en los paquetes de Swing es grande y no podrían cubrirse todos en este libro. De hecho, para cubrir por completo Swing se requeriría todo un libro. Sin embargo, estos dos capítulos proporcionan los conocimientos básicos de este importante tema.

NOTA *Más información de Swing puede ser consultada en mi libro "Swing: A Beginner's Guide" publicado por McGraw-Hill / Osborne (2007).*

Los orígenes de Swing

Swing no existía en las primeras versiones de Java. Swing surgió como respuesta a las deficiencias presentadas por el subsistema original de interfaces gráficas en Java: AWT. AWT define un conjunto básico de controles, ventanas y cuadros de diálogo que soportan una útil pero limitada interfaz gráfica. Una razón para la naturaleza limitada de la AWT es que ésta traduce sus diversos componentes en los equivalentes en la plataforma específica. Eso significa que la apariencia del componente está definida por la plataforma y no por Java. Debido a que los componentes de AWT utilizan recursos del código nativo, estos componentes son denominados de *peso completo*.

El uso de componentes nativos causa diversos problemas. Primero, debido a las variantes entre los sistemas operativos un componente puede verse o actuar diferente en diferentes plataformas. Esta variabilidad potencial amenaza la filosofía de Java: escríbela uno vez, ejecútalo en cualquier parte. Segundo, la apariencia de cada componente era inalterable (debido a que estaba definida por la plataforma) y no podía ser (fácilmente) cambiada. Tercero, el uso de componentes de peso completo causaba algunas restricciones frustrantes. Por ejemplo, un componente de peso completo siempre es rectangular y opaco.

Poco tiempo después de la aparición de la versión original de Java, resultó obvio que las limitaciones y restricciones presentes en AWT eran suficientemente serias y se requería una mejor solución, esta solución fue Swing. Swing fue incluido en 1997 como parte del Java Foundation Classes (JFC). Swing estuvo inicialmente disponible en Java 1.1 como una biblioteca independiente. Sin embargo, a partir de Java 1.2, Swing (y el resto del JFC) fue completamente integrada a Java.

Swing está construido sobre AWT

Antes de avanzar, es necesario remarcar un aspecto importante: Swing elimina diversas limitaciones inherentes a AWT pero Swing *no* remplaza a AWT. En lugar de eso, Swing está construido sobre las bases de AWT. Éste es el motivo por el cual AWT aún es una parte crucial de Java. Swing utiliza el mismo mecanismo de gestión de eventos de AWT. Por lo tanto, conocer AWT y el modelo de manejo de eventos es necesario para comprender a Swing. AWT se examina en los Capítulos 23 y 24. El mecanismo de gestión de eventos se describe en el Capítulo 22.

Dos características clave de Swing

Como se explicó antes, Swing fue creado para superar las limitaciones presentes en AWT. Esto se logra a través de dos características clave: componentes ligeros y una apariencia configurable. Juntas, estas características proporcionan una solución elegante y sencilla para los problemas de AWT. Más que ninguna otra cosa, estas dos características definen la esencia de Swing. Cada una se examina a continuación.

Los componentes de Swing son ligeros

Salvo pocas excepciones, los componentes de Swing son *ligeros*. Esto significa que son escritos completamente en Java y no se proyectan directamente en un elemento correspondiente en la plataforma específica. Debido a que los componentes ligeros se dibujan utilizando gráficas primitivas, estos pueden ser transparentes, lo cual permite utilizar formas no rectangulares. Por ello, los componentes ligeros son más eficientes y flexibles. Además, debido a que los componentes ligeros no se transforman en componentes nativos, su apariencia está determinada por Swing, no por el sistema operativo subyacente. Esto significa que cada componente trabajará de manera consistente en todas las plataformas.

La apariencia de un componente es independiente del componente mismo

Swing soporta una *apariencia configurable* llamada PLAF (por las siglas en inglés de Pluggable Look And Feel). Debido a que cada componente de Swing se dibuja utilizando código de Java y no componentes nativos, la apariencia de un componente está bajo el control de Swing. Esto significa que es posible separar la apariencia del componente de su lógica, esto es lo que Swing hace. Separar la apariencia proporciona una ventaja significativa: hace posible cambiar la forma en que un componente es dibujado sin afectar ningún aspecto adicional. En otras palabras, es posible "conectar" una nueva apariencia a cualquier componente sin crear ningún efecto secundario en el código que utiliza dicho componente. Más aún, es posible definir conjuntos completos de apariencias que representan a diferentes estilos de interfaces gráficas. Para utilizar un estilo

específico, su apariencia se "conecta" a la aplicación. Una vez hecho esto, todos los componentes se redibujan automáticamente utilizando el estilo especificado.

La apariencia configurable ofrece diversas ventajas. Es posible definir una apariencia consistente para todas las plataformas. Por el contrario, es posible crear una apariencia que sea similar a la que se ve en una plataforma específica. Por ejemplo, si sabemos que una aplicación será ejecutada sólo en un ambiente Windows, es posible especificar la apariencia de Windows. También es posible diseñar una apariencia personalizada. Finalmente, la apariencia puede ser alterada dinámicamente durante la ejecución del programa.

Java SE 6 proporciona apariencias como metal y Motif, que están disponibles para todos los usuarios de Swing. La apariencia metal también se denomina la *apariencia de Java*. Esta apariencia es independiente de la plataforma y está disponible para todos los ambientes de ejecución. Es también la apariencia por omisión. Los ambientes Windows tienen acceso además a la apariencia Windows y Windows Classic. Este libro utiliza la apariencia por omisión (metal) debido a que es independiente de la plataforma.

El modelo MVC

En general, un componente visual es una composición de tres aspectos diferentes:

- La forma en que el componente luce cuando es dibujado en pantalla.
- La forma en que el componente reacciona ante el usuario.
- La información de estado asociada al componente.

Sin importar la arquitectura que se utilice para implementar un componente, éste debe contener implícitamente estas tres partes. Por años, una arquitectura de componentes ha probado ser excepcionalmente efectiva: la arquitectura de *Modelo–Vista–Controlador*, o simplemente MVC.

La arquitectura MVC es exitosa debido a que cada pieza de diseño corresponde a un aspecto del componente. En la terminología de MVC, el *modelo* corresponde a la información del estado asociado con el componente. Por ejemplo, en el caso de un checkbox, el modelo contiene un campo que indica si la caja está activa o inactiva. La vista determina cómo el componente se muestra en pantalla, incluyendo cualquier aspecto que sea relativo al estado del modelo. El *controlador* determina cómo reacciona el componente ante el usuario. Por ejemplo, cuando el usuario hace clic sobre un checkbox, el controlador reacciona cambiando al modelo para reflejar la selección del usuario (un checkbox activo o inactivo). Luego, esto ocasiona que se actualice la vista. Separando un componente en un modelo, una vista y un controlador, permite cambiar la implementación específica de cada parte sin afectar a las otras dos partes. Por ejemplo, diferentes implementaciones de vista pueden dibujar al mismo componente en diferentes formas sin afectar al modelo o al controlador.

Aunque la arquitectura MVC y los principios detrás de ella son conceptualmente sanos, el alto nivel de separación entre la vista y el controlador no es benéfico para los componentes de Swing. En lugar de ello, Swing utiliza una versión modificada de MVC la cual combina la vista y el controlador en una sola entidad lógica denominada *comisionado UI*. Por esta razón la estrategia de Swing es llamada arquitectura *Modelo–Comisionado* o arquitectura de *Modelo a parte*. Por ello, aunque la arquitectura de los componentes de Swing está basada en MVC, ésta no utiliza una implementación clásica de esta arquitectura.

La apariencia configurable de Swing es posible debido a su arquitectura Modelo-Comisionado. Debido a que la vista y el controlador están separados del modelo, la apariencia

puede cambiar sin afectar el cómo se utiliza al componente dentro del programa. Además, es posible personalizar al modelo sin afectar la forma en que el componente aparece en pantalla o responde a las entradas del usuario.

Para soportar la arquitectura Modelo-Comisionado, la mayoría de los componentes de Swing contienen dos objetos. El primero representa al modelo. El segundo representa al comisionado. El modelo está definido por interfaces. Por ejemplo, el modelo para un botón está definido por la interfaz **ButtonModel**. Los comisionados son clases que heredan de **ComponentUI**. Por ejemplo, el comisionado para un botón es **ButtonUI**. Normalmente nuestros programas no van a interactuar directamente con los comisionados.

Componentes y contenedores

Una interfaz gráfica de Swing consiste en dos elementos clave: *componente* y *contenedor*. Sin embargo, esta distinción es principalmente conceptual debido a que todos los contenedores también son componentes. La diferencia entre los dos se encuentra en su propósito: como el término se emplea comúnmente, un *componente* es un control visual independiente, como un botón o un deslizador. Un contenedor posee a un grupo de componentes. Por ello, un contenedor es un tipo especial de componente que está diseñado para poseer a otros componentes. Además, para que un componente sea desplegado debe ser colocado en un contenedor. Por ello, todas las interfaces gráficas hechas con Swing contienen al menos un contenedor. Debido a que los contenedores son componentes, un contenedor puede también poseer a otros contendores. Esto le permite a Swing definir lo que se denomina una *contención jerárquica*, en cuyo nivel más alto se encuentra lo que se denomina un *contenedor raíz*.

Veamos de cerca los componentes y contenedores.

Componentes

En general los componentes de Swing se derivan de la clase **JComponent**. Las únicas excepciones a esto son los cuatro contenedores raíz que se describen en la siguiente sección. La clase **JComponent** proporciona la funcionalidad común a todos los componentes. Por ejemplo, la clase **JComponent** soporta la apariencia configurable. La clase **JComponent** hereda de las clases **Container** y **Component**. Por ello, un componente de Swing está construido sobre componentes AWT y es además compatible con componentes AWT.

Todos los componentes de Swing están representados por clases definidas en el paquete **javax.swing**. La siguiente tabla muestra los nombres de las clases para los componentes de Swing (incluyendo aquellos que se utilizan como contenedores).

JApplet	JButton	JCheckBox	JCheckBoxMenuItem
JColorChooser	JComboBox	JComponent	JDesktopPane
JDialog	JEditorPane	JFileChooser	JFormattedTextField
Jframe	JInternalFrame	JLabel	JLayeredPane
JList	JMenu	JMenuBar	JMenuItem
JOptionPane	JPanel	JPasswordField	JPopupMenu
JProgressBar	JRadioButton	JRadioButtonMenuItem	JRootPane
JScrollBar	JScrollPane	JSeparator	JSlider

JSpinner	JSplitPane	LTabbedPane	JTable
JTextArea	JTextField	JTextPane	JTogglebutton
JToolBar	JToolTip	JTree	JViewport
JWindow			

Observe que todas las clases comienzan con la letra **J**. Por ejemplo, la clase para crear una etiqueta es **JLabel**; la clase para un botón es **JButton**; y la clase para un scrollbar es **JScrollBar**.

Contenedores

Swing define dos tipos de contenedores. El primero son los contenedores raíz: **JFrame**, **JApplet**, **JWindow** y **JDialog**. Estos contenedores no heredan de **JComponent**. Sin embargo, heredan de las clases **Component** y **Container** definidas en AWT. A diferencia de los otros componentes de Swing que se consideran ligeros, los contenedores raíz son componentes pesados. Esto hace de los contenedores raíz un caso especial en la biblioteca de componentes de Swing.

Como el nombre lo indica un contenedor raíz debe estar en lo alto de una jerarquía de contención. Un contenedor raíz no está contenido en ningún otro componente. Además, todas las jerarquías de contención deben comenzar con un contenedor raíz. El contenedor utilizado más comúnmente en aplicaciones es **JFrame**. El contenedor utilizado para applets es **JApplet**.

El segundo tipo de contenedores soportados por Swing son los contenedores ligeros. Los contenedores ligeros heredan de la clase **JComponent**. Un ejemplo de contenedor ligero es **JPanel**, el cual es un contenedor de propósito general. Los contenedores ligeros se utilizan a menudo para organizar y administrar grupos de componentes relacionados debido a que un contenedor ligero puede ser contenido por otros contenedores. Así, el programador puede utilizar contenedores ligeros como **JPanel** para crear subgrupos de controles relacionados que están contenidos en un contenedor exterior.

Los contenedores raíz

Cada contenedor raíz define un conjunto de espacios. A la cabeza de la jerarquía se encuentra una instancia de **JRootPane**. **JRootPane** es un contenedor ligero cuyo propósito es administrar a los otros espacios. También ayudan a administrar la barra de menú opcional. Los espacios que abarca el espacio raíz se llaman *glass pane*, *content pane* y *layered pane*.

El glass pane es el espacio de mayor nivel. Se localiza encima del resto de los espacios y los cubre completamente. Por omisión, ésta es una instancia transparente de **JPanel**. El glass pane nos permite por ejemplo, administrar eventos del ratón que afectan a todo el contenedor (y no a un control individual) o pintar sobre cualquier otro componente. En muchos casos, el programador no necesitará utilizar el glass pane directamente, pero ahí está cuando se necesite emplearlo.

El layered pane es una instancia de **JLayeredPane**. El layered pane permite asignar un valor de profundidad a los componentes. Este valor determina cual componente cubre a otro. Así, el layered pane nos permite especificar un orden en Z para un componente, a pesar de que esto no es algo que se requiera usualmente. El layered pane contiene al content pane y opcionalmente a una barra de menú.

Aunque el glass pane y los layered pane están integrados a la operación de un contenedor raíz y sirven a propósitos importantes, mucho de lo que proveen ocurre detrás de la escena. El espacio con el cual interactúa la aplicación principalmente es el content pane, debido a ello

este es el espacio al cual se añaden elementos visuales. En otras palabras, cuando se añade un componente, como un botón, al contenedor raíz, éste se añadirá en el content pane. Por omisión, el content pane es una instancia opaca de **JPanel**.

Los paquetes de Swing

Swing es un subsistema realmente grande que hace uso de muchos paquetes. Éstos son los paquetes utilizados por Swing que están definidos por Java SE 6.

java.swing	java.swing.border	java.swing.colorchooser
java.swing.event	java.swing.filechooser	java.swing.plaf
java.swing.plaf.basic	java.swing.plaf.metal	java.swing.plaf.multi
java.swing.plaf.synth	java.swing.table	java.swing.text
java.swing.text.html	java.swing.text.html.parser	java.swing.text.rtf
java.swing.tree	java.swing.undo	

El paquete principal es **java.swing**. Este paquete debe ser importado en cualquier programa que utiliza Swing. Este contiene las clases que implementan los componentes fundamentales de Swing, como botones, etiquetas y checkbox.

Una aplicación sencilla con Swing

Los programas con Swing son diferentes de los programas con salida a consola y los programas con AWT mostrados antes en este libro. Por ejemplo, utilizan un conjunto de componentes y una jerarquía de contenedores diferentes a las de AWT. Los programas con Swing además tienen requerimientos especiales relativos al manejo de hilos. La mejor forma de entender la estructura de un programa en Swing es ver un ejemplo. Existen dos tipos de programas en Java en los cuales se emplea Swing. El primero son las aplicaciones de escritorio. El segundo son los applets. En está sección mostramos como crear una aplicación con Swing. La creación de applets con Swing se describe más adelante en este mismo capítulo.

Aunque un poco pequeño, el siguiente programa muestra una forma de escribir una aplicación con Swing.Y se muestran al mismo tiempo varias características claves de Swing. El ejemplo utiliza dos componentes de Swing: **JFrame** y **JLabel**. **JFrame** es el contenedor raíz que se emplea comúnmente en las aplicaciones Swing. **JLabel** es el componente de Swing que crea una etiqueta, la cual es un componente que se emplea para desplegar información. La etiqueta es el componente más simple de Swing debido a su pasividad. Esto es, una etiqueta no responde a las entradas del usuario. Una etiqueta sólo despliega información. El programa utiliza un contenedor **JFrame** para sujetar una instancia de **JLabel**. La etiqueta muestra un pequeño mensaje de texto.

```
// Una aplicación sencilla con Swing

import javax.swing.*;

class SwingDemo {

  SwingDemo() {
```

```
    // crear un nuevo contenedor JFrame
    JFrame jfrm = new JFrame ("Una aplicación simple en Swing");

    // definir un tamaño inicial
    jfrm.setSize(275, 100);

    // finalizar la ejecución del programa cuando el usuario cierra la aplica-
    ción
    jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    // crear una etiqueta con texto
    JLabel jlab = new JLabel ("Swing permite crear excelente interfaces gráfi-
    cas");

    // añadir la etiqueta al content pane
    jfrm.add(jlab);

    // desplegar el contenedor
    jfrm.setVisible(true);
  }

  public static void main(String args[]) {
    // crear el contenedor y el hilo de gestión de eventos
    SwingUtilities.invokeLater(new Runnable() {
      public void run() {
        new SwingDemo();
      }
    });
  }
}
```

Los programas con Swing se compilan y ejecutan de la misma forma que las otras aplicaciones de Java. Para compilar el programa anterior, se puede emplear la siguiente línea de comando:

```
javac SwingDemo.java
```

Para ejecutar el programa, se utiliza la siguiente línea de comando:

```
java SwingDemo
```

Cuando el programa se ejecuta, se genera la ventana mostrada en la Figura 29-1.

Debido a que el programa anterior ilustra varios conceptos fundamentales de Swing, vamos a examinarlo cuidadosamente línea por línea. El programa comienza importando **javax.swing**. Como se mencionó este paquete contiene los componentes y modelos definidos por Swing. Por ejemplo **java.swing** define las clases que implementan etiquetas, botones, controles de texto y menús. Este paquete debe ser incluido en cualquier programa que utilice Swing.

En seguida, el programa declara la clase **SwingDemo** y un constructor para la clase. El constructor es el lugar donde gran parte de la acción del programa ocurre. El constructor comienza creando un **JFrame**, utilizando la siguiente línea de código:

```
JFrame jfrm = new JFrame ("Una aplicación simple en Swing");
```

Ésta crea un contenedor llamado **jfrm** que define una ventana rectangular completa con una barra de título; botones para cerrar, minimizar, maximizar y reabrir; así como un menú de sistema. Esto es, el contenedor crea una ventana estándar de alto nivel. El título de la ventana se define en el constructor.

FIGURA 29-1 La ventana producida por el programa **SwingDemo**

En seguida, la ventana es dimensionada utilizando la sentencia:

```
jfrm.setSize(275, 100);
```

El método **setSize( )** (heredado por **JFrame** de la clase **Component** de AWT) establece las dimensiones de la ventana, las cuales se especifican en píxeles. La forma general del método es la siguiente:

void setSize(int *ancho*, int *alto*);

En el ejemplo, el ancho de la ventana es 275 y la altura es 100.

Por omisión cuando una ventana raíz se cierra (por ejemplo cuando el usuario hace clic en el botón de cierre), la ventana se elimina de la pantalla, pero la aplicación no termina. Mientras que el comportamiento por omisión es útil para algunas situaciones, no es lo que se necesita para la mayoría de las aplicaciones. En lugar de ello, el programador usualmente deseará que la aplicación completa finalice cuando su ventana raíz sea cerrada. Existen un par de formas para lograr esto. La forma más fácil es llamando al método **setDefaultCloseOperation( )**, como en el programa anterior:

```
jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

Después de ejecutarse la sentencia anterior, cerrar la ventana causará que la aplicación completa termine. La forma general del método **setDefaultCloseOperation( )** se muestra a continuación:

void setDefaultCloseOperation(int *w*)

El valor dado en *w* determina lo que ocurre cuando la ventana se cierra. Existen varias operaciones que se pueden indicar en w además de **JFrame.EXIT_ON_CLOSE.** Las operaciones válidas son:

JFrame.DISPOSE_ON_CLOSE

JFrame.HIDE_ON_CLOSE

JFrame.DO_NOTHING_ON_CLOSE

Estos nombres son un reflejo de las acciones realizadas. Las constantes correspondientes están declaradas en la interfaz **WindowsConstants**, la cual es una interfaz declarada en **javax.swing** que es implementada por **JFrame**.

La siguiente línea de código crea un componente de tipo **JLabel**:

```
JLabel jlab = new JLabel ("Swing permite crear excelente interfaces gráficas");
```

JLabel es el componente más simple y fácil de utilizar debido a que no acepta entradas del usuario. Este componente simplemente despliega información, la cual puede consistir en texto, un icono, o una combinación de ambos. La etiqueta creada por el programa contiene sólo texto, el cual se coloca como argumento para el constructor.

La siguiente línea de código añade la etiqueta en el content pane:

```
jfrm.add(jlab);
```

Como se explicó antes, todos los contenedores raíz tienen un content pane en el cual los componentes se almacenan. Por ello, para añadir un componente, éste se añade al content pane del contenedor. Esto se logra llamando al método **add( )** sobre la referencia a **JFrame** (**jfrm** en este caso). La forma general del método **add( )** es ésta:

Component add(Component *c*)

El método **add( )** es heredado por **JFrame** de la clase **Container** del AWT.

Por omisión, el content pane asociado con **JFrame** utiliza un acomodo de los elementos en el borde. La versión de **add( )** mostrada antes añade la etiqueta al centro del contenedor. Otras versiones de **add( )** permiten especificar otra de las regiones en los bordes. Cuando un componente se añade al centro, su tamaño se ajusta automáticamente para llenar el espacio al centro.

Antes de continuar, un punto histórico importante necesita ser mencionado. Antes de JDK 5, cuando se añadía un componente al content pane, no se podía invocar al método **add( )** directamente sobre la instancia de **JFrame**. En lugar de ello, era necesario llamar al método **add( )** en el content pane del objeto **JFrame**. El content pane puede obtenerse llamando al método **getContentPane( )** sobre una instancia de **JFrame**. El método **getContentPane( )** se muestra a continuación:

Container **getContentPane( )**

Este método regresa una referencia de tipo Container al content pane. El método **add( )** es luego llamado sobre esta referencia para añadir un componente al content pane. Así, en el pasado, era necesario utilizar la siguiente sentencia para añadir el objeto **jlab** en el objeto **jfrm**:

```
jfrm.getContentPane().add(jlab); // estilo antiguo
```

Aquí, el método **getContentPane** primero obtiene una referencia al content pane, y luego el método **add( )** añade el componente al contenedor enlazado a este espacio. Este mismo procedimiento fue también necesario para la invocación del método **remove( )** para eliminar un componente y para invocar al método **setLayout( )** para establecer el administrador de diseño en el content pane. Veremos llamadas explicitas al método **getContentPane( )** frecuentemente a lo largo de todo el código anterior a la versión 5.0 de Java. Hoy día, el uso del método **getContentPane( )** no es necesario. Podemos simplemente llamar a **add( )**, **remove( )** y **setLayout( )** directamente sobre **JFrame** debido a que estos métodos han sido modificados para que funcionen sobre el content pane directamente.

La última sentencia en el constructor **SwingDemo** causa que la ventana sea visible:

```
jfrm.setVisible(true);
```

El método **setVisible( )** es heredado de la clase **Component** de AWT. Si este argumento tiene el valor **true** la ventana es desplegada. En caso contrario, estará oculta. Por omisión, un **JFrame** es invisible, así que el método **setVisible(true)** debe ser llamado para mostrarlo.

Dentro de **main( )**, se crea un objeto de tipo **SwingDemo**, el cual causa que la ventana y la etiqueta sean mostrados en pantalla. Observe que el constructor **SwingDemo** se invoca utilizando las siguientes líneas de código:

```
SwingUtilities.invokeLater(new Runnable() {
```

```
  public void run() {
    new SwingDemo();
  }
});
```

Esta secuencia causa que un objeto de tipo **SwingDemo** sea creado en el *hilo responsable de despachar* los eventos en lugar de crearse en el hilo principal de la aplicación. Esto debido a que los programas de Swing son aplicaciones orientadas a eventos. Por ejemplo, cuando el usuario interactúa con un componente, un evento se genera. Un evento se pasa a la aplicación llamando a un gestor de eventos definido por la aplicación. Sin embargo, el gestor se ejecuta en el hilo despachador de eventos proporcionado por Swing y no en el hilo principal de la aplicación. Así, aunque los gestores de eventos se definen por el programa, estos son llamados en un hilo que no ha sido creado por el programa.

Para evitar problemas (incluida la posibilidad de bloqueos de tipo deadlock), todos los componentes gráficos de Swing deben ser creados y actualizados en el hilo gestor de eventos no en el hilo principal de la aplicación. Sin embargo el método **main( )** sí es ejecutado en el hilo principal. Así que el método **main( )** no puede instanciar directamente un objeto **SwingDemo**. En lugar de ello, debe crear un objeto de tipo **Runnable** que ejecute al hilo despachador de eventos y que además sea dicho hilo el responsable de crear a la interfaz gráfica.

Para permitir que el código de la interfaz gráfica sea creado en el hilo despachador de eventos, se debe utilizar uno de los dos métodos definidos por la clase **SwingUtilities**. Estos métodos son **invokeLater( )** e **invokeAndWait( )**. Estos métodos se muestran a continuación:

```
static void invokeLater(Runnable obj)

static void invokeAndWait(Runnable obj)
  throws InterruptedException, InvocationTargetException
```

Donde, *obj* es un objeto **Runnable** que tendrá su propio método **run( )** que será llamado por el hilo despachador de eventos. La diferencia entre los dos métodos es que **invokeLater( )** regresa inmediatamente, pero **invokeAndWait( )** espera hasta que **obj.run( )** regresa. El programador puede utilizar uno de estos métodos para llamar a un método que construya la interfaz gráfica para su aplicación Swing, o cuando necesite modificar el estado de la interfaz gráfica desde un código no ejecutado por el hilo despachador de eventos. Normalmente utilizaremos **invokeLater( )**, como lo hace el programa anterior. Sin embargo, cuando se construye la interfaz gráfica inicial de un applet, será necesario utilizar el método **invokeAndWait( )**.

Gestión de eventos

El ejemplo anterior mostró la forma básica de un programa con Swing, pero omitió una parte importante: la gestión de eventos. Debido a que **JLabel** no recibe entradas del usuario, no genera eventos, por tanto no se requirió de un gestor de eventos. Sin embargo, otros componentes de Swing responden a las entradas del usuario y los eventos producidos por estas interacciones con el usuario deben ser gestionados. Los eventos además pueden ser generados en formas que no están directamente relacionadas con el usuario. Por ejemplo, un evento se genera cuando un cronometro alcanza su valor final. Cualquiera que sea el caso, la gestión de eventos abarca una gran parte de cualquier aplicación basada en Swing.

El mecanismo de gestión de eventos utilizado por Swing es el mismo que el utilizado por AWT. Esta estrategia se llama modelo de *delegación de eventos* y se describe en el Capítulo 22. En muchos casos, Swing utiliza los mismos eventos que AWT, y estos eventos están empaquetados en **java.awt.event**. Los eventos específicos de Swing están empaquetados en **javax.swing.event**.

Aunque los eventos son gestionados en Swing en la misma forma que son gestionados con AWT, aún es importante ver un ejemplo. El siguiente programa gestiona los eventos generados por un botón de Swing. Un ejemplo de la salida del programa se muestra a continuación en la Figura 29-2.

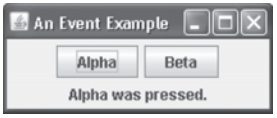


FIGURA 29-2 Salida producida por el programa **EventDemo**


```
// Gestión de eventos en un programa con Swing

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class EventDemo {

  JLabel jlab;

  EventDemo() {

    // crear un nuevo contenedor JFrame
    JFrame jfrm = new JFrame ("Ejemplo con eventos");

    // especificar FlowLayout como el administrador de diseño
    jfrm.setLayout(new FlowLayout());

    // definir un tamaño inicial
    jfrm.setSize(220, 90);

    // finalizar la ejecución del programa cuando el usuario cierra la aplicación
    jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    // construir dos botones
    JButton jbtnAlfa = new JButton("Alfa");
    JButton jbtnBeta = new JButton("Beta");

    // añade listener al botón alfa
    jbtnAlfa.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent ae) {
        jlab.setText("Alfa fue presionado.");
      }
    });

    // añade listener al botón beta
    jbtnAlfa.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent ae) {
    jlab.setText("Beta fue presionado.");
    }
    });
```

```
    // añade los botones al content pane
    jfrm.add(jbtnAlfa);
    jfrm.add(jbtnBeta);

    // crea una etiqueta con texto
    jlab = new JLabel("Presione un botón");

    // añade la etiqueta al content pane
    jfrm.add(jlab);

    // desplegar el contenedor
    jfrm.setVisible(true);
  }

  public static void main(String args[]) {
    // crear el contenedor y el hilo de gestión de eventos
    SwingUtilities.invokeLater(new Runnable() {
      public void run() {
        new EventDemo();
      }
    });
  }
}
```

Primero, observe que el programa ahora importa los paquetes **java.awt** y **java.awt.event.** El paquete **java.awt** es necesario debido a que contiene a la clase **FlowLayout**, la cual brinda el soporte necesario para utilizar el administrador de diseño estándar de tipo flujo para acomodar los componentes en el contenedor. Véase el Capítulo 24 para obtener mayores referencias a los administradores de diseño. El paquete **java.awt.event** se requiere porque éste define la interfaz **ActionListener** y la clase **ActionEvent**.

El constructor **EventDemo** comienza creando un objeto de tipo **JFrame** llamado **jfrm**. Luego establece un objeto **FlowLayout** como el administrador de diseño para el content pane asociado a **jfrm**. Recuerde que por omisión, el content pane utiliza un objeto de tipo **BorderLayout** como administrador de diseño. Sin embargo, para este ejemplo, un objeto **FlowLayout** es más apropiado. Nótese que un **FlowLayout** se asigna utilizando la siguiente sentencia:

```
jfrm.setLayout(new FlowLayout());
```

Como se explicó antes, en el pasado era necesario llamar explícitamente al método **getContentPane( )** para establecer el administrador de diseño para el content pane. Esta acción se eliminó a partir de JDK 5.

Después de establecer el tamaño y la operación de cierre por omisión, el constructor **EventDemo( )** crea dos botones, como se muestra a continuación:

```
JButton jbtnAlfa = new JButton("Alfa");
JButton jbtnBeta = new JButton("Beta");
```

El primer botón contendrá el texto "Alfa" y el segundo el texto "Beta". Los botones de Swing son instancias de **JButton**. **JButton** proporciona varios constructores. El constructor utilizado aquí es

JButton(String *msg*)

El parámetro *msg* especifica la cadena que será desplegada dentro del botón.

Cuando un botón es presionado, éste genera un evento de tipo **ActionEvent**. Por ello, **JButton** proporciona el método **addActionListener( )**, el cual es utilizado para añadir un listener de acción. **JButton** también proporciona el método **removeActionListener( )** para eliminar un

listener, este método no se utiliza en el ejemplo anterior. Como se explicó en el capítulo 22, la interfaz **ActionListener** define sólo un método: **actionPerformed( )**. El cual se muestra a continuación:

void actionPerformed(ActionEvent *ae*)

Este método es invocado cuando se presiona un botón. En otras palabras, es el gestor de evento que es llamado cuando ocurre un evento sobre el botón.

A continuación, se añaden los listener para eventos de acción a los botones, con el siguiente código:

```
// añade listener al botón alfa
jbtnAlfa.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent ae) {
    jlab.setText("Alfa fue presionado.");
  }
});

// añade listener al botón beta
jbtnAlfa.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent ae) {
    jlab.setText("Beta fue presionado.");
  }
});
```

Aquí se utilizan clases interiores anónimas para definir los gestores de evento de los botones. Cada vez que se presiona un botón, la cadena mostrada en el objeto **jlab** cambia para reflejar cuál fue el botón presionado.

Luego, los botones son añadidos al content pane del objeto jfrm:

```
jfrm.add(jbtnAlfa);
jfrm.add(jbtnBeta);
```

Finalmente, **jlab** se añade al content pane y la ventana se hace visible. Cuando se ejecuta el programa, cada vez que se presione un botón, un mensaje se muestra en la etiqueta, el mensaje indica cual botón fue presionado.

Un último comentario: recuerde que todos los gestores de eventos, como el método **actionPerformed( )**, son llamados en el hilo despachador de eventos. Por consiguiente, un gestor de eventos debe regresar rápidamente a fin de evitar disminuir la velocidad de ejecución de la aplicación. Si la aplicación requiere hacer algo que consuma mucho tiempo como resultado de un evento, deberá utilizar un hilo aparte.

Crear un applet con Swing

El segundo tipo de programa que comúnmente utiliza Swing es el applet. Los applet basados en Swing son similares a los applets basados en AWT, pero con una diferencia importante: un applet hecho con Swing hereda de la clase **JApplet** en lugar de heredar de la clase **Applet**. La clase **JApplet** es una clase derivada de **Applet**. Por ello, **JApplet** incluye toda la funcionalidad de la clase **Applet** y añade soporte para Swing. **JApplet** es un contenedor de alto nivel en Swing, lo cual significa que no es una derivación de **JComponent**. Debido a que **JApplet** es un contenedor de alto nivel o raíz, éste incluye los espacios descritos antes. Esto significa que todos los componentes se añaden al content pane de **JApplet** de la misma forma que los componentes se añaden al content pane de un **JFrame**.

Los applet de Swing utilizan el mismo ciclo de vida con los cuatro métodos descritos en el capítulo 21: **init( )**, **start( )**, **stop( )** y **destroy( )**. Por supuesto, necesitamos sobrescribir sólo aquellos métodos que son requeridos por el applet. Pintar en un applet de Swing es diferente a pintar en un applet AWT, además un applet de Swing normalmente no sobrescribe al método **paint( )**. Cómo pintar en Swing se describe más adelante en este capítulo.

Una nota adicional: toda la interacción con los componentes en un applet de Swing debe tener lugar en el hilo despachador de eventos como se describió en la sección anterior. Esta cuestión relativa al manejo de hilos aplica a todos los programas hechos con Swing.

FIGURA 29-3 Salida generada por el ejemplo de applet con Swing

A continuación un ejemplo de applet hecho con Swing. Este ejemplo realiza las mismas acciones que la aplicación previa. La Figura 29-3 muestra el programa al ser ejecutado por **appletviewer.**

```
// Un applet sencillo con Swing

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

/*
Este es el código HTML necesario para ejecutar al applet:

<object code="MySwingApplet" width=220 height=90>
</object>
*/

public class MySwingApplet extends JApplet {
  JButton jbtnAlfa;
  JButton jbtnBeta;

  JLabel jlab;

  // inicializar al applet
  public void init() {
    try {
      SwingUtilities.invokeAndWait (new Runnable() {
        public void run() {
          makeGUI(); //inicializa la interfaz gráfica
        }
      });
    } catch (Exception exc) {
      System.out.println("No es posible inicializar al applet debido a " + exc);
    }
  }
```

```
  // este applet no requiere sobrescribir start(), stop() o destroy()

  // inicializar la interfaz gráfica
  private void makeGUI() {
    // coloca un FlowLayout en el applet
    setLayout(new FlowLayout());

    // construir dos botones
    JButton jbtnAlfa = new JButton("Alfa");
    JButton jbtnBeta = new JButton("Beta");

    // añade listener al botón alfa
    jbtnAlfa.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent ae) {
        jlab.setText("Alfa fue presionado.");
      }
    });

    // añade listener al botón beta
    jbtnAlfa.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent ae) {
        jlab.setText("Beta fue presionado.");
      }
    });

    // añade los botones al content pane
    jfrm.add(jbtnAlfa);
    jfrm.add(jbtnBeta);

    // crea una etiqueta con texto
    jlab = new JLabel("Presione un botón");

    // añade la etiqueta al content pane
    jfrm.add(jlab);
  }
}
```

Hay dos cosas importantes que debemos observar en el applet anterior. Primero, **MySwingApplet** hereda de **JApplet**. Como se explicó antes, todos los applets basados en Swing heredan de **JApplet** en lugar de heredar de **Applet**. Segundo, el método **init( )** inicializa a los componentes Swing en el hilo despachador de eventos realizando una llamada al método **makeGUI( )**. Nótese que esto se logra utilizando al método **invokeAndWait( )** en lugar de **invokeLater( )**. Los applets deben utilizar **invokeAndWait( )** debido a que el método **init( )** no debe regresar sino hasta que el proceso completo de inicialización haya sido completado. En esencia el método **start( )** no puede ser invocado sino hasta después de la inicialización, lo cual significa que la interfaz gráfica debe estar completamente construida.

Dentro del método **makeGUI( )**, los dos botones y la etiqueta son creados, y los listener de acción son añadidos a los botones. Finalmente, los componentes son añadidos al content pane. Aunque este ejemplo es un poco simple, ésta es la misma estrategia general que debe ser utilizada cuando se construye cualquier interfaz gráfica en Swing dentro de un applet.

Dibujar en Swing

Aunque el conjunto de componentes de Swing es muy poderoso, el programador no está limitado a sólo utilizarlo, Swing permite al programador escribir directamente en el área de

trabajo de un contenedor e incluso de un componente. Aunque el uso de Swing *no* siempre requiere de dibujar directamente en la superficie de un componente, esta acción está disponible para aquellas aplicaciones que requieran de esta habilidad.

Para escribir directamente en la superficie de un componente, se utilizan los métodos de dibujo definidos por AWT, como **drawLine( )** o **drawRect( )**. Así, muchas de las técnicas y métodos descritos en el Capítulo 23 funcionan en Swing. Sin embargo, existen además algunas diferencias importantes, y el proceso se discute a detalle en esta sección.

Fundamentos de dibujo

La estrategia de Swing para dibujar está basada en el mecanismo original de AWT, pero la implementación de Swing ofrece un control más fino. Antes de examinar los detalles específicos del dibujo en Swing, es útil revisar el mecanismo utilizado por AWT que lo fundamenta.

La clase **Component** de AWT define un método llamado **paint( )** que se emplea para dibujar directamente en la superficie de un componente. Para la mayoría de los componentes, el método **paint( )** no es llamado por el programa. De hecho sólo en los casos más inusuales será este método llamado por el programa. En lugar de ello, el método **paint( )** es llamado por el sistema de ejecución de Java siempre que un componente debe ser dibujado. Esta situación puede ocurrir debido a diversas razones. Por ejemplo, la ventana en la cual el componente es desplegado puede quedar cubierta por otra ventana y luego ser descubierta. O bien, la ventana puede ser minimizada y luego reactivada. El método **paint( )** también se llama cuando un programa comienza su ejecución. Cuando se escribe código basado en AWT, una aplicación deberá sobrescribir a **paint( )** si necesita escribir directamente en la superficie de un componente.

Debido a que **JComponent** hereda de **Component**, todos los componentes ligeros de Swing heredan al método **paint( )**. Sin embargo, no se debe sobrescribir el método para pintar directamente en la superficie del componente. La razón es que Swing utiliza una estrategia un poco más sofisticada para dibujar, la cual involucra a tres diferentes métodos: **paintComponent( )**, **paintBorder( )** y **paintChildren( )**.

Estos métodos dibujan la porción indicada de un componente y dividen el proceso de dibujo en tres diferentes acciones lógicas. En un componente ligero, el método **paint( )** original de AWT simplemente ejecuta llamadas a los tres métodos en el orden descrito.

Para pintar en la superficie de un componente Swing, debemos crear una subclase del componente y luego sobrescribir su método **paintComponent( )**. Éste es el método que pinta el interior de un componente. El programador normalmente no sobrescribirá los otros dos métodos de dibujo. Cuando se sobrescribe al método **paintComponent( )**, la primera cosa que se debe hacer es llamar a **super.paintComponent( )**, para que la porción de la superclase correspondiente al proceso de dibujo se realice. El único momento en que esto no es necesario es cuando se toma un control manual completo sobre como un componente se despliega. Después de esto se escribe la salida deseada. El método **paintComponent( )** se muestra a continuación:

```
protected void paintComponent(Graphics g)
```

El parámetro *g* representa al contexto gráfico en el cual se escribirá la salida.

Para provocar que un componente sea pintado bajo el control del programa, se llama al método **repaint( )**. Esto funciona en Swing justo como en AWT. El método **repaint( )** está definido por la

clase **Component**. Llamarlo causa que el sistema llame a **paint( )** tan pronto como sea posible hacerlo. Debido a que pintar es una operación que consume tiempo, este mecanismo permite al sistema de ejecución de Java aplazar la operación de dibujo momentáneamente, por ejemplo, hasta que otra tarea de mayor prioridad sea completada. Por supuesto, en Swing la llamada al método **paint( )** produce una llamada al método **paintComponent( )**. Por consiguiente, para dibujar en la superficie de un componente, nuestro programa deberá almacenar la salida hasta que **paintComponent( )** sea invocado. Dentro del método **paintComponent( )** sobrescrito, debemos dibujar la salida almacenada por el programa.

Calcular el área de dibujo

Cuando se dibuja sobre la superficie de un componente, debemos ser cuidadosos para restringir la salida en el área que esta dentro del borde. Aunque Swing automáticamente recorta cualquier salida que exceda los límites de un componente, aún es posible pintar en los bordes, lo cual se corrige cuando el borde es dibujado. Para evitar esto, se debe calcular el *área de dibujo* del componente. Esta es el área definida por el tamaño actual del componente menos el espacio utilizado por el borde. Por consiguiente, antes de pintar en un componente, debemos obtener el tamaño del borde y luego ajustar el dibujo.

Para obtener el tamaño del borde, se llama al método **getInsets( )**, mostrado a continuación:

```
Insets getInsets( )
```

Este método está definido en la clase **Container** y sobrescrito por **JComponent**. El método regresa un objeto **Insets** que contiene las dimensiones del borde. Los valores del objeto Insets pueden ser accedidos leyendo los campos:

```
int top;
int bottom;
int left;
int right;
```

Estos valores se utilizan para calcular el área de dibujo conociendo el ancho y largo del componente. Se puede obtener el ancho y largo del componente llamando a los métodos **getWidth( )** y **getHeight( )**. Estos métodos se muestran a continuación:

```
int getWidth( )
int getHeight( )
```

Restando los valores del objeto Insets al ancho y largo se puede calcular el tamaño del área útil para dibujo.

Un ejemplo con dibujos

El siguiente programa pone en práctica los conceptos anteriores. El programa crea una clase llamada **PaintPanel** que hereda de **JPanel**. El programa luego utiliza un objeto de esta clase para mostrar líneas cuyos extremos han sido generados al azar. Un ejemplo de la ejecución del programa se muestra en la Figura 29-4.

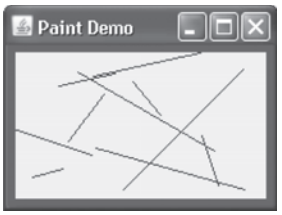


FIGURA 29-4 Ejemplo de la salida del programa **PaintPanel**

```
// dibujar líneas en un panel

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.util.*;

// esta clase hereda de JPanel. La clase sobrescribe
// al método paintComponent() para que se dibujen
// líneas aleatorias en el panel
class PaintPanel extends JPanel {
  Insets ins; // almacena la dimensión de los bordes

  Random rand; // se utiliza para generar números aleatorios

  // crea un panel
  PaintPanel() {

    // coloca un borde alrededor del panel
    setBorder(BorderFactory.createLineBorder(Color.RED, 5));

    rand = new Random();
  }

  // sobrescribe el método paintComponent()
  protected void paintComponent(Graphics g) {
    // siempre se debe llamar al método de la superclase
    super.paintComponent(g);

    int x, y, x2, y2;

    // se obtiene el largo y ancho del componente
    int alto = getHeight();
    int ancho = getWidth();

    // obtener los valores de los bordes
    ins = getInsets();

    // dibujar diez líneas cuyos extremos son generados aleatoriamente
    for(int i=0; i<10; i++) {
      // obtener valores aleatorios para definir
      // los extremos de cada línea
      x = rand.nextInt(ancho - ins.left);
      y = rand.nextInt(alto - ins.bottom);
      x2 = rand.nextInt(ancho - ins.left);
      y2 = rand.nextInt(alto - ins.bottom);
```

```
      // dibujar la línea
      g.drawLine(x, y, x2, y2);
    }
  }
}

// ejemplo de dibujo en un panel
class paintDemo {

  Jlabel jlab;
  PaintPanel pp;

  PaintDemo() {

    // crea un nuevo contenedor
    JFrame jfrm = new JFrame("Ejemplo de dibujo en componentes");

    // establecer el tamaño inicial
    jfrm.setSize(200, 150);

    // el programa termina cuando el usuario cierra la aplicación
    jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    // crea el panel donde se va a dibujar
    pp = new PaintPanel();

    // añade el panel al content pane. Debido a que se utiliza el administrador
    // de diseño por omisión (border), el panel se coloca automáticamente al
    // centro del contenedor
    jfrm.add(pp);

    // muestra el contenedor
    jfrm.setVisible(true);
  }

  public static void main(String args[]) {
    // crea el contenedor y el hilo de gestión de eventos
    SwingUtilities.invokeLater(new Runnable() {
      public void run() {
        new PaintDemo();
      }
    });
  }
}
```

Examinemos el programa a detalle. La clase **paintPanel** hereda de **JPanel**. **JPanel** es uno de los contenedores ligeros de Swing, lo cual significa que es un componente que puede ser añadido al espacio de content pane de un objeto tipo **JFrame**. Para gestionar el dibujo, **PaintPanel** sobrescribe el método **paintComponent( )**. Esto le permite a **PaintPanel** escribir directamente en la superficie del componente. El tamaño del panel no se especifica debido a que el programa emplea el administrador de diseño por omisión el cual agrega el panel al centro con un tamaño que lo hace ocupar toda el área de trabajo. Si se cambia el tamaño de la ventana, el tamaño del panel se ajustará en la misma proporción.

Nótese que el constructor también especifica un borde en color rojo con 5 píxeles de ancho. Esto es realizado por el método **setBorder( )**, mostrado a continuación:

void setBorder(Border *border*)

Border es la interfaz de Swing que encapsula a un borde. Se puede obtener un borde llamando a uno de los métodos de fábrica definidos en la clase **BorderFactory**. El método utilizado en el programa anterior es **createLineBorder( )**, el cual crea un borde sencillo. La forma general del método es:

static Border createLineBorder(Color *color*, int *ancho*)

Aquí, *color* especifica el color del borde y *ancho* especifica el ancho en píxeles.

Dentro del método sobrescrito **paintComponent( )**, observe que la primera acción que realiza es llamar a **super.paintComponent( )**. Como se explicó antes, esto es necesario para asegurarnos de que el componente sea dibujado adecuadamente. Luego se obtiene el ancho y alto del panel así como de sus bordes. Estos valores se utilizan para asegurar que las líneas se sitúan dentro del área de dibujo del panel. El área de dibujo tiene un ancho y largo igual al ancho y largo del componente menos el ancho del borde. Los cálculos están diseñados para trabajar con diferentes tamaños de bordes y componentes. Para comprobar esto, podemos intentar cambiar el tamaño de la ventana. Las líneas permanecerán situadas dentro de los bordes del panel.

La clase **PaintDemo** crea un objeto **PaintPanel** y luego añade el panel en el espacio correspondiente al content pane del contenedor. Cuando la aplicación se muestra por primera vez, el método **paintComponent** sobrescrito es llamado, y luego las líneas se dibujan. Cada vez que se redimensiona o se oculta y restaura la ventana, un nuevo conjunto de líneas se dibuja. En todos los casos, las líneas se trazan en el área de dibujo.

30 CAPÍTULO

Explorando Swing

En el capítulo anterior se describieron muchos de los conceptos principales relativos a Swing y se mostraron de forma general tanto las aplicaciones como los applets con Swing. Este capítulo continúa la discusión de Swing presentando una vista general de varios de los componentes de Swing, como son botones, checkbox, árboles y tablas. Los componentes Swing proveen una rica funcionalidad y permiten un gran nivel de personalización. Debido a las limitaciones de espacio, no es posible describir todas sus características y atributos. En lugar de eso, el propósito de esta sección será mostrar una visión general de las capacidades del conjunto de componentes de Swing.

Las clases que definen a los componentes de Swing que se describen en este capítulo son:

JButton	JCheckBox	JComboBox	JLabel
JList	JRadioButton	JScrollPanel	JTabbedPane
JTable	JTextField	JToggleButton	JTree

Todos estos componentes son ligeros, lo cual significa que todos son derivados de **JComponent**.

También se discutirá la clase **ButtonGroup**, la cual encapsula un conjunto de botones de Swing mutuamente excluyentes, e **ImagenIcon**, la cual encapsula una imagen gráfica. Ambas son definidas por Swing y empaquetadas en **javax.swing**.

Un comentario adicional: los componentes Swing son mostrados utilizando applets debido a que el código de los applets es más compacto que el código de las aplicaciones de escritorio. Sin embargo, las mismas técnicas aplican para ambos, tanto aplicaciones como applets.

JLabel e ImageIcon

JLabel es un componente fácil de utilizar. Este componente crea una etiqueta y fue introducido en el capítulo anterior. Aquí, veremos a **JLabel** un poco más de cerca. **JLabel** puede ser utilizado para mostrar texto y/o un icono. Es un componente pasivo y no responde a una entrada de usuario. **JLabel** define varios constructores. A continuación se listan tres de ellos:

JLabel(Icon *icon*)
JLabel(String *str*)
JLabel(String *str*, Icon *icon*, int *alinear*)

Donde, *str* e *icon* son el texto e icono utilizados para la etiqueta. El argumento *alinear* específica la alineación horizontal del texto y/o icono dentro de las dimensiones de la etiqueta. Debe ser uno de los siguientes valores: **LEFT**, **RIGHT**, **CENTER**, **LEADING**, o **TRAILING**. Estas constantes están definidas en la interfaz **SwingConstants**, junto con otros valores utilizados por las clases Swing.

Note que los iconos están especificados por objetos de tipo **Icon**, la cual es una interfaz definida por Swing. La forma más sencilla de obtener un icono es utilizar la clase **ImagenIcon**. La clase **ImagenIcon** implementa a la interfaz **Icon** y encapsula una imagen. De esta manera, un objeto de tipo **ImageIcon** puede ser pasado como argumento al parámetro **Icon** del constructor de **JLabel**. Existen muchas formas de proveer la imagen, incluyendo leerla de un archivo o descargarla de un URL. A continuación un ejemplo del constructor utilizado para el ejemplo de esta sección:

ImageIcon(String *archivo*)

Obtiene la imagen del archivo llamado *archivo*.

El icono y el texto asociados con una etiqueta pueden ser obtenidos por los siguientes métodos:

Icon getIcon( )
String getText( )

El icono y el texto asociado con la etiqueta pueden ser modificados por los siguientes métodos:

void setIcon(Icon *icon*)
void setText(String *str*)

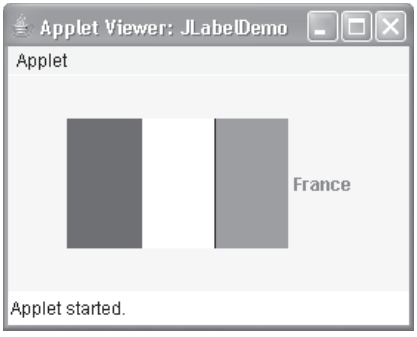
Donde, *icon* y *str* son el icono y el texto respectivamente. Además, usando **setText( )** es posible cambiar el texto dentro de la etiqueta durante la ejecución del programa.

El siguiente applet ilustra cómo crear y mostrar una etiqueta que contiene ambos, un icono y una cadena. Comienza creando un objeto **ImagenIcon** para el archivo **francia.gif**, el cual representa la bandera de Francia. Este objeto se utiliza como segundo argumento para el constructor de **JLabel**. El primer y último argumento para el constructor de **JLabel** son el texto de la etiqueta y la alineación. Finalmente, la etiqueta se agrega al contenedor principal.

```
// Ejemplo de JLabel e ImageIcon
import java.awt.*;
import javax.swing.*;
/*
  <applet code="JLabelDemo" width=250 height=150>
  </applet>
*/

public class JLabelDemo extends JApplet {

  public void init() {
    try {
      SwingUtilities.invokeAndWait (
        new Runnable() {
          public void run() {
            makeGUI() ;
          }
        }
      );
```

```
    } catch (Exception exc) {
      System.out.println("No se puede crear la aplicación debido a: " + exc);
    }
  }

  private void makeGUI() {

    // Se crea un icono
    ImageIcon ii = new ImageIcon("francia.gif");

    // Se crea la etiqueta
    JLabel jl = new JLabel ("Francia", ii, JLabel.CENTER);

    // Agrega la etiqueta
    add(jl);
  }
}
```

La salida de este ejemplo se muestra a continuación:



JTextField

JTextField es el componente más simple de Swing. Es también, probablemente, el componente de texto más ampliamente utilizado. **JTextField** permite editar una línea de texto. Este componente se deriva de **JTextComponent**, el cual provee la funcionalidad básica común a todos los componentes de texto de Swing. **JTextField** utiliza la interfaz **Document** en su modelo.

Tres de los constructores de **JTextField** se muestran a continuación:

JTextField (int *cols*)
JTextField (String *str*, int *cols*)
JTextField (String *str*)

Donde *str* es la cadena que será mostrada inicialmente, y *cols* es el número de columnas en el campo de texto especificado. Si no se especifica una cadena, el campo de texto estará inicialmente vacío. Si el número de columnas no está especificado, el campo de texto será del tamaño justo para que la cadena dada se ajuste.

JTextField genera eventos en respuesta a la interacción del usuario. Por ejemplo, un **ActionEvent** se dispara cuando el usuario presiona ENTER. Un evento de tipo **CaretEvent** se genera cada vez que el cursor cambia de posición (**CaretEvent** está empaquetado en **javax.**

swing.event). Existen además otros eventos. En muchos casos, los programas no necesitarán el manejo de eventos. En su lugar, regularmente se obtiene la cadena actual del campo de texto cuando se necesita. Para obtener el texto presente en el campo de texto, se utiliza el método **getText( )**.

El siguiente ejemplo ilustra el uso de **JTextField**. El programa crea un **JTextField** y lo agrega al contenedor principal. Cuando el usuario presiona ENTER, se genera un **ActionEvent**. Esto causa que el texto escrito en el campo de texto sea mostrado en la barra de estado de la ventana.

```
// Ejemplo de JTextField.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
  <applet code="JTextFieldDemo" width=300 height=50>
  </applet>
*/

public class JTextFieldDemo extends JApplet {
  JTextField jtf;

  public void init() {
    try {
      SwingUtilities.invokeAndWait (
        new Runnable() {
          public void run() {
            makeGUI();
          }
      }
    );
  } catch (Exception exc) {
    System.out.println("No se puede crear la aplicación debido a: " + exc);
  }
}

private void makeGUI() {

    // Cambia el organizador a FlowLayout.
    setLayout(new FlowLayout());

    // Agrega el campo de texto al contenedor principal
    jtf = new JTextField(15);
    add(jtf);
    jtf.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent ae) {
        // Muestra el texto cuando el usuario presiona ENTER.
        showStatus(jtf.getText());
      }
    });
  }
}
```

La salida del ejemplo se muestra a continuación:

Los botones de Swing

Swing define cuatro tipos de botones: **JButton**, **JToggleButton**, **JCheckBox** y **JRadioButton**. Todos son subclases de la clase abstracta **AbstractButton**, la cual extiende de **JComponent**. De esta manera, todos los botones comparten un conjunto común de atributos.

La clase **AbstractButton** contiene varios métodos que permiten el control del comportamiento de los botones. Por ejemplo, se pueden definir diferentes iconos para ser mostrados por el botón cuando se deshabilita, se presiona, o selecciona. Otro icono puede ser utilizado como un icono de *traslado*, el cual se muestra cuando el ratón se posiciona sobre un botón. Los siguientes métodos definen dichos iconos:

```
void setDisabledIcon(Icon di)
void setPressedIcon(Icon pi)
void setSelectedIcon(Icon si)
void setRolloverIcon(Icon ri)
```

Donde, *di*, *pi*, *si* y *ri* son los iconos a ser utilizados para el propósito indicado por el nombre (en inglés) de cada método.

El texto asociado con un botón puede ser leído y modificado vía los siguientes métodos:

```
String getText( )
void setText(String str)
```

Donde, *str* es el texto que será asociado con el botón.

El modelo utilizado por todos los botones está definido por la interfaz **ButtonModel**. Un botón genera un **ActionEvent** cuando se presiona. Existen además otros eventos. A continuación se examina cada una de las clases concretas para botones.

JButton

La clase **JButton** provee la funcionalidad de un botón clásico. Un ejemplo de esta clase de botón se vio ya en el capítulo anterior. La clase **JButton** permite que un icono, una cadena o ambos sean asociados con el botón. Tres de sus constructores se muestran a continuación:

```
JButton(Icon icon)
JButton(String str)
JButton(String str, Icon icon)
```

Donde *str* e *icon* son la cadena y el icono utilizado para el botón.

Cuando se presiona el botón, se genera un evento de tipo **ActionEvent**. El objeto **ActionEvent** se pasa al método **actionPerformed( )** del **ActionListener** registrado, y se utiliza para obtener la cadena del comando de acción asociado con el botón. Por omisión, la cadena de comando de acción es la misma que se muestra dentro del botón. Sin embargo, se puede definir un *comando de acción* llamando al método **setActionCommand( )** del botón. También se puede obtener el comando de acción llamando al método **getActionCommand( )** del objeto que representa al evento. Este método está declarado como sigue:

```
String getActionCommand( )
```

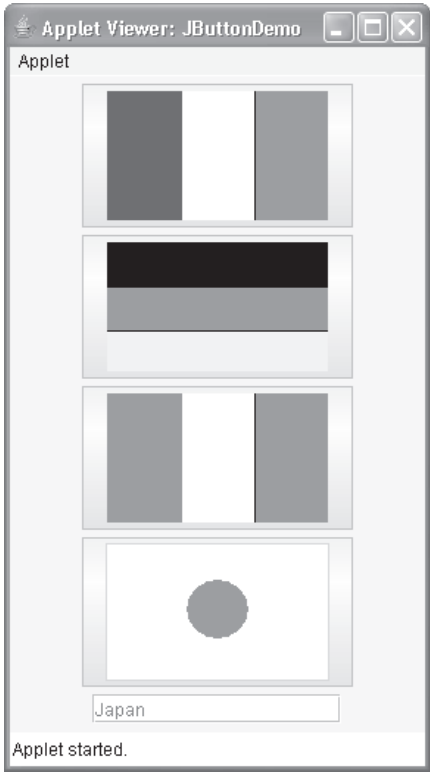
El comando de acción identifica al botón. De está forma, cuando se utilizan dos o más botones dentro de la misma aplicación, el comando de acción nos proporciona una forma fácil de determinar cuál botón fue presionado.

En el capítulo anterior, se vio un ejemplo de un botón basado en texto. El siguiente ejemplo muestra un ejemplo de botón con icono. El ejemplo muestra cuatro botones y una etiqueta. Cada botón muestra un icono que representa la bandera de un país. Cuando se presiona un botón, el nombre del país se muestra en la etiqueta.

```
// Ejemplo de botones con iconos.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
  <applet code="JButtonDemo" width=250 height=450>
  </applet>
*/

public class JButtonDemo extends JApplet
implements ActionListener {
  JLabel jlab;

  public void init() {
    try {
      SwingUtilities.invokeAndWait(
        new Runnable() {
          public void run() {
            makeGUI();
          }
        }
      );
    } catch (Exception exc) {
      System.out.println("No se puede crear la aplicación debido a: " + exc);
    }
  }

private void makeGUI() {

    // cambia a FlowLayout.
    setLayout(new FlowLayout());

    // Agrega los botones
    ImageIcon francia = new ImageIcon("francia.gif");
    JButton jb = new JButton(francia);
    jb.setActionCommand("Francia");
    jb.addActionListener(this);
    add(jb);

    ImageIcon alemania = new ImageIcon("alemania.gif");
    jb = new JButton(alemania);
    jb.setActionCommand("Alemania");
    jb.addActionListener(this);
    add(jb);

    ImageIcon italia = new ImageIcon("italia.gif");
    jb = new JButton(italia);
    jb.setActionCommand("Italia");
    jb.addActionListener(this);
    add(jb);
```

```
    ImageIcon japon = new ImageIcon("japon.gif");
    jb = new JButton(japon);
    jb.setActionCommand("Japón");
    jb.addActionListener (this);
    add(jb);

    //Crea y agrega la etiqueta
    jlab = new JLabel ("Seleccione una bandera");
    add(jlab);
  }

  //Gestiona los eventos de los botones
  public void actionPerformed(ActionEvent ae) {
    jlab.setText("Seleccionó" + ae.getActionCommand());
  }
}
```

La salida del programa se muestra aquí:



JToggleButton

Una variación útil de un botón clásico es el llamado *botón interruptor*. Un botón interruptor parece un botón clásico, pero actúa diferente, porque sólo tiene dos estados: presionado y liberado. Esto es, cuando se presiona un botón interruptor, éste permanece presionado, en lugar de regresar a su posición original como un botón regular lo hace. Cuando se presiona el botón interruptor por segunda ocasión, se libera. Por lo tanto, cada vez que un botón interruptor es presionado, alterna entre sus dos estados.

Los botones interruptores son objetos de la clase **JToggleButton**. La clase **JToggleButton** implementa a **AbstractButton**. La clase **JToggleButton** además de ser la clase de donde se generan los botones interruptor, es la superclase de otros dos componentes de Swing que también representan un control de dos estados. Éstos son los **JCheckBox** y **JRadioButton**, los cuales son descritos posteriormente en este capítulo. Así, **JToggleButton** define la funcionalidad básica de todos los componentes de dos estados.

JToggleButton define varios constructores. El constructor utilizado por los ejemplos de esta sección es:

JToggleButton(String *str*)

Este constructor crea un botón interruptor que contiene el texto dado en *str*. Por omisión, el botón está en estado de apagado. Otros constructores permiten la creación de botones interruptor que contienen imágenes o imágenes y texto.

La clase **JToggleButton** utiliza un modelo definido por una clase anidada llamada **JToggleButton**.**ToggleButtonModel**. Normalmente, no necesitará interactuar directamente con el modelo para utilizar al botón interruptor estándar.

Así como **JButton**, **JToggleButton** genera un **ActionEvent** cada vez que es presionado. A diferencia de **JButton**, sin embargo, **JToggleButton** también genera un evento de tipo elemento. Este evento es utilizado por los componentes que soportan el concepto de selección. Cuando un **JToggleButton** se presiona y queda hacia adentro, éste se selecciona. Cuando se presiona y esto causa que quede hacia afuera, se deselecciona.

Para gestionar los eventos de tipo elemento, se debe implementar la interfaz **ItemListener**. Recuerde del Capítulo 22, que cada vez que un evento de tipo elemento es generado, éste se pasa al método **itemStateChanged( )** definido por **ItemListener**. Dentro del método **itemStateChanged( ),** el método **getItem( )** puede ser llamado sobre el objeto **ItemEvent** para obtener una referencia a la instancia de **JToggleButton** que generó el evento. Este método se muestra a continuación:

```
Object getItem( )
```

Se devuelve una referencia al botón. Se necesitará hacer una conversión de esta referencia a **JToggleButton**.

La forma más sencilla de determinar el estado de un botón interruptor es llamando al método **isSelected( )** (heredado de **AbstractButton**) sobre el botón que generó el evento, como se muestra a continuación:

```
boolean isSelected( )
```

El método devuelve **verdadero** si el botón está seleccionado y **falso** en caso contrario.

A continuación se muestra un ejemplo que utiliza un botón interruptor. Observe cómo trabaja el listener de tipo elemento. El listener simplemente llama al método **isSelected( )** para determinar el estado del botón.

```
// Ejemplo de JToggleButton.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
  <applet code="JToggleButtonDemo" width=200 height=80>
  </applet>
*/

public class JToggleButtonDemo extends JApplet {

  JLabel jlab;
  JToggleButton jtbn;

  public void init() {
    try {
      SwingUtilities.invokeAndWait(
        new Runnable() {
          public void run() {
            makeGUI();
          }
        }
```

```
      );
    } catch (Exception exc) {
      System.out.println("No se puede crear la aplicación debido a: " + exc);
    }
  }

  private void makeGUI() {

    // Cambia a FlowLayout.
    setLayout(new FlowLayout());

    // Crea una etiqueta.
    jlab = new JLabel("El botón está apagado");

    // Crea un botón interruptor.
    jtbn = new JToggleButton("Encendido/Apagado");

    // Agrega un ItemListener al botón interruptor
    jtbn.addItemListener(new ItemListener() {
      public void itemStateChanged(ItemEvent ie) {
        if(jtbn.isSelected())
          jlab.setText("El botón está encendido .");
        else
          jlab.setText("El botón está apagado");
      }

    });

    // Agrega el botón interruptor y la etiqueta
    add(jtbn);
    add(jlab);
  }
}
```

La salida del ejemplo anterior se muestra a continuación:

JCheckBox

La clase **JCheckBox** provee la funcionalidad de un checkbox. Su superclase inmediata es **JToggleButton**, la cuál provee soporte para los dos estados del botón, justo como se describió antes. La clase **JCheckBox** define varios constructores. El que se utilizará aquí es:

JCheckBox(String *str*)

El cual crea un checkbox que contiene el texto especificado por *str* como etiqueta. Otros constructores permiten especificar el estado de selección inicial del botón y especificar un icono.

Cuando el usuario selecciona o deselecciona un check box, se genera un **ItemEvent**. Se puede obtener una referencia al **JCheckBox** que generó el evento llamando al método **getItem( )** del objeto **ItemEvent** enviado como parámetro al método **itemStateChange( )** definido en **ItemListener**. La forma más sencilla de determinar el estado de un checkbox es llamando al método **isSelected( )** sobre la instancia **JCheckBox**.

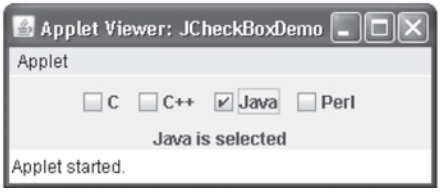
Además de soportar la operación normal del checkbox, la clase **JCheckBox** permite especificar el icono que indicará que el checkbox está seleccionado, libre o en la mira. No se utilizará esa capacidad aquí, pero está disponible para ser utilizada en nuestros programas.

El siguiente ejemplo ilustra el uso de checkbox. Muestra cuatro checkbox y una etiqueta. Cuando el usuario hace clic en un checkbox, se genera un **ItemEvent**. Dentro el método **itemStateChanged( )**, se llama al método **getItem( )** para obtener una referencia al objeto **JCheckBox** que generó el evento. A continuación se llamada al método **isSelected( )** para determinar si el checkbox fue seleccionado o limpiado. El método **getText( )** obtiene el texto del checkbox y lo utiliza para definir el texto dentro de la etiqueta.

```
// Ejemplo de JCheckbox.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
  <applet code="JCheckBoxDemo" width=270 height=50>
  </applet>
*/

public class JCheckBoxDemo extends JApplet
implements ItemListener {
  JLabel jlab;

  public void init() {
    try {
      SwingUtilities.invokeAndWait(
        new Runnable() {
          public void run() {
            makeGUI();
          }
        }
      );
    } catch (Exception exc) {
      System.out.println("No se puede crear la aplicación debido a: " + exc);
    }
  }

  private void makeGUI() {

    // Cambia a FlowLayout.
    setLayout(new FlowLayout());

    // Agrega los checkbox
    JCheckBox cb = new JCheckBox("C");
    cb.addItemListener (this);
    add(cb);

    cb = new JCheckBox("C++");
    cb.addItemListener(this);
    add(cb);
```

```
    cb = new JCheckBox("Java");
    cb.addItemListener(this);
    add(cb);

    cb = new JCheckBox("Perl") ;
    cb.addItemListener(this);
    add(cb);

    // Crea la etiqueta y la agrega
    jlab = new JLabel("Seleccione lenguajes");
    add(jlab);
  }

  // Gestor de eventos
  public void itemStateChanged(ItemEvent ie) {
    JCheckBox cb = (JCheckBox)ie.getItem();

    if(cb.isSelected())
      jlab.setText (cb.getText () + " está seleccionado");
    else
      jlab.setText(cb.getText() + " no está seleccionado")
  }
}
```

La salida del ejemplo se muestra aquí:



JRadioButton

Los botones de radio son un grupo de botones mutuamente excluyentes, en los cuales solamente un botón puede estar seleccionado al mismo tiempo. Estos botones son soportados por la clase **JRadioButton**, la cual extiende de **JToggleButton**. **JRadioButton** provee varios constructores. El constructor utilizado en el ejemplo se muestra aquí:

JRadioButton(String *str*)

Donde *str* es la etiqueta para el botón. Otros constructores permiten especificar el estado inicial de la selección y especificar un icono.

Para que la exclusión mutua sea activada, los botones de radio deben ser configurados dentro de un grupo. Sólo uno de los botones en el grupo puede ser seleccionado al mismo tiempo. Por ejemplo, si un usuario presiona un botón de radio que está en un grupo, cualquier selección previamente seleccionada en el mismo grupo es deseleccionada automáticamente. Un grupo de botones se crea con la clase **ButtonGroup**. Su constructor por omisión se invocado para este propósito. Los elementos son luego agregados a un grupo vía el siguiente método:

void add(AbstractButton *ab*)

Aquí, *ab* es una referencia al botón que será agregado al grupo.

Un **JRadioButton** genera eventos de acción, eventos de elemento y eventos de cambio cada vez que se selecciona un botón. Frecuentemente, el evento de acción es el que gestiona, lo que significa que normalmente implementará la interfaz **ActionListener**.

Recuerde que el único método definido por **ActionListener** es **actionPerformed( )**. Dentro de este método, se pueden utilizar diferentes formas para determinar cuál botón fue seleccionado. Primero, se puede revisar el comando de acción llamando al método **getActionCommand( )**. Por omisión, el comando de acción es el mismo que la etiqueta del botón, pero se puede fijar el comando de acción a un valor diferente llamando al método **setActionCommand( )** del botón radio. Segundo, se puede llamar al método **getSource( )** del objeto **ActionEvent** y comparar la referencia con el botón. Finalmente, se puede simplemente revisar cada botón de radio para encontrar cuál está seleccionado actualmente llamando al método **isSelected( )** de cada botón. Recuerde, cada vez que ocurre un evento de acción, significa que el botón que ha sido seleccionado tiene cambios y que uno y sólo un botón será seleccionado.

El siguiente ejemplo ilustra como utilizar botones de grupo. Se crean tres botones de tipo radio. Los botones son entonces agregados al grupo. Como se explicó, esto es necesario para mantener el comportamiento mutuamente exclusivo. Al presionar un botón de radio se genera un evento de acción, el cual es gestionado por el método **actionPerformed( )**. Como parte de la gestión, el método **getActionCommand( )** obtiene el texto que está asociado con el botón radio y lo utiliza para definir el texto dentro de la etiqueta.

```
// Ejemplo de JRadioButton
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
  <applet code="JRadioButtonDemo" width=300 height=50>
  </applet>
*/

public class JRadioButtonDemo extends JApplet
implements ActionListener {
  JLabel jlab;

  public void init() {
    try {
      SwingUtilities.invokeAndWait(
        new Runnable () {
          public void run() {
            makeGUI();
          }
        }
      );
    } catch (Exception exc) {
      System.out.println("No se puede crear la aplicación debido a: " + exc);
    }
  }

  private void makeGUI() {

    // Cambia a FlowLayout.
    setLayout(new FlowLayout());

    // Crea los botones radio y los agrega
    JRadioButton bl = new JRadioButton ("A");
    bl.addActionListener(this);
    add(b1);
```

```
    JRadioButton b2 = new JRadioButton("B");
    b2.addActionListener(this);
    add(b2);

    JRadioButton b3 = new JRadioButton("C") ;
    b3.addActionListener(this);
    add(b3);

    // Define un grupo de botones
    ButtonGroup bg = new ButtonGroup() ;
    bg.add(bl);
    bg.add(b2);
    bg.add(b3);

    // Crea una etiqueta y la agrega.
    jlab = new JLabel("Seleccione uno");
    add(jlab);
  }

  // Gestiona la selección de botones.
  public void actionPerformed(ActionEvent ae) {
    jlab.setText("Ha seleccionado " + ae.getActionCornmand());
  }
}
```

La salida del ejemplo se muestra a continuación:

JTabbedPane

La clase **JTabbedPane** encapsula a un *cuadro tabulado*. Éste administra a un grupo de componentes ligándolos con tabuladores. El seleccionar un tabulador causa que el componente asociado con él sea colocado en primer plano en el cuadro. Los **JTabbedPane** son componentes comunes en las interfaces gráficas modernas. Dada la compleja naturaleza de los **JTabbedPane**, es sorprendente la facilidad con que se pueden crear y utilizar.

La clase **JTabbedPane** define tres constructores. Se utilizará el constructor por omisión, el cual crea un control vació con los tabuladores posicionados a lo largo de la parte superior del cuadro. Los otros dos constructores permiten especificar la posición de los tabuladores, los cuales pueden estar en cualquiera de los cuatro lados. La clase **JTabbedPane** utiliza el modelo **SingleSelectionModel**.

Los tabuladores son agregados llamando al método **addTab( )**. A continuación se muestra una de sus formas:

void addTab(String *nom*, Component *comp*)

Donde *nom* es el nombre de un tabulador y *comp* es el componente que deberá ser agregado al tabulador. Frecuentemente, el componente agregado en un tabulador es un **JPanel**, el cual

contiene a un grupo de componentes relacionados. Esta técnica permite que un tabulador contenga a un grupo de componentes.

El procedimiento general para utilizar un cuadro tabulado es el siguiente:

1. Crear una instancia de la clase **JTabbedPane**.
2. Añadir cada tabulador llamando al método **addTab( )**.
3. Añadir el cuadro tabulado al contenedor principal.

El siguiente ejemplo ilustra el uso de un JTabbedPane. El primer tabulador se titula "Ciudades" y contiene cuatro botones. Cada botón muestra el nombre de una ciudad. El segundo tabulador se titula "Colores" y contiene tres checkbox. Cada checkbox muestra el nombre de un color. El tercer tabulador se titula "Sabores" y contiene una lista. Esto permite al usuario seleccionar uno de tres sabores.

```
// Ejemplo de JTabbedPane.
import javax.swing.*;
/*
  <applet code="JTabbedPaneDemo" width=400 height=100>
  </applet>
*/

public class JTabbedPaneDemo extends JApplet {

  public void init() {
    try {
      SwingUtilities.invokeAndWait(
        new Runnable() {
          public void run() {
            makeGUI();
          }
        }
      );
    } catch (Exception exc) {
      System.out.println("No se puede crear la aplicación debido a: " + exc);
    }
  }

  private void makeGUI() {

    JTabbedPane jtp = new JTabbedPane();
    jtp.addTab("Ciudades", new PanelCiudades());
    jtp.addTab("Colores", new PanelColores());
    jtp.addTab("Sabores", new PanelSabores());
    add(jtp);
  }
}

// Crea los paneles que serán agregados al cuadro tabulado.
class PanelCiudades extends JPanel {

  public PanelCiudades() {
    JButton b1 = new JButton("Nueva York");
    add(b1);
    JButton b2 = new JButton("Londres");
    add(b2);
    JButton b3 = new JButton("Hong Kong");
```

```
    add(b3);
    JButton b4 = new JButton ("Tokio");
    add(b4);
  }
}

class PanelColores extends JPanel {

  public PanelColores() {
    JCheckBox cbl = new JCheckBox ("Rojo");
    add(cbl);
    JCheckBox cb2 = new JCheckBox("Verde");
    add(cb2);
    JCheckBox cb3 = new JCheckBox("Azul") ;
    add(cb3);
  }
}

class PanelSabores extends JPanel {

  public PanelSabores() {
    JComboBox jcb = new JComboBox() ;
    jcb.addItem("Vainilla");
    jcb.addItem ("Chocolate");
    jcb.addItem ("Fresa");
    add(jcb);
  }
}
```

La salida del ejemplo se muestra en las siguientes tres ilustraciones:

JScrollPane

JScrollPane es un contenedor ligero que automáticamente gestiona el desplazamiento de otros componentes. El componente que está siendo desplazado puede ser un componente individual,

tal como una tabla, o un grupo de componentes contenidos dentro de otro contenedor ligero tal como un **JPanel**. En cualquier caso, si el objeto que está siendo desplazado es más grande que el área visible, se generan automáticamente barras en horizontal y/o vertical y el componente podrá ser desplazado. Dado que **JScrollPane** automáticamente desplaza a su contenido, este componente elimina la necesidad de manejar de forma independiente a las barras de desplazamiento.

El área visible de un área de desplazamiento se denomina *viewport*. Ésta es la ventana en la cual el componente que está siendo desplazado se muestra. El viewport muestra la porción visible del componente contenido en el **JScrollPane**. Las barras de desplazamiento mueven al componente dentro del viewport. Por omisión, un **JScrollPane** añadirá o eliminará las barras de desplazamiento dinámicamente conforme sea necesario. Por ejemplo, si el componente es más alto que el viewport entonces se agregará una barra de desplazamiento vertical. Si el componente se ajusta completamente al viewport entonces se eliminan las barras de desplazamiento.

La clase **JScrollPane** define varios constructores. El constructor que utilizaremos en los ejemplos de este capítulo es el siguiente:

JScrollPane(Componente *comp*)

El componente a ser desplazado está especificado por *comp*. Las barras de desplazamiento son mostradas automáticamente cuando el contenido del cuadro excede la dimensión del viewport.

Éstos son los pasos para seguir para utilizar un JScrollPane:

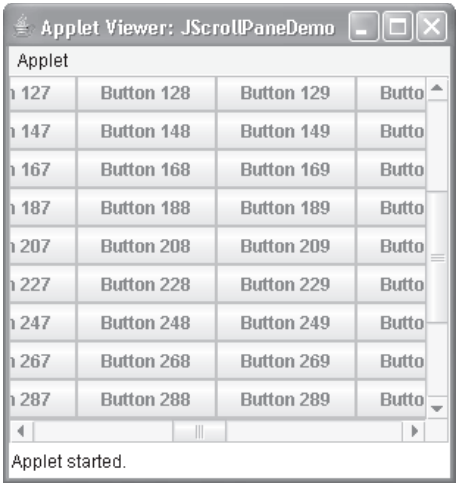
1. Crear el componente a ser desplazado.
2. Crea una instancia de **JScrollPane**, pasándole el objeto a ser desplazado.
3. Agregar el objeto **JScrollPane** al contenedor principal.

El siguiente ejemplo ilustra el uso de un **JScrollPane**. Primero, se crea un objeto **JPanel**, y se agregan 400 botones en él, organizados en 20 columnas. Luego, este panel es agregado al **JScrollPane**, y el **JScrollPane** es agregado al contenedor principal. Puesto que el panel es más grande que el viewport, aparecen automáticamente barras de desplazamiento vertical y horizontal. Se puede utilizar las barras de desplazamiento para mover los botones dentro de la vista.

```
// Ejemplo de JScrollPane.
import java.awt.*;
import javax.swing.*;
/*
  <applet code="JScrollPaneDemo" width=300 height=250>
  </applet>
*/

public class JScrollPaneDemo extends JApplet {

  public void init() {
    try {
      SwingUtilities.invokeAndWait(
        new Runnable() {
          public void run() {
            makeGUI();
          }
        }
```

```
      );
    } catch (Exception exc) {
      System.out.println("No se puede crear la aplicación debido a: " + exc);
    }
  }
  private void makeGUI() {

    // Agrega 400 botones al panel
    JPanel jp = new JPanel();
    jp.setLayout(new GridLayout(20,20));
    int b = 0;
    for (int i = 0; i<20; i++){
      for (int j = 0; j<20; j++){
        jp.add (new JButton ("Botón" + b));
        ++b;
      }
    }

    // Crea el JScrollPane
    JScrollPane jsp = new JScrollPane(jp);

    //Agrega el JScrollPane al contenedor principal
    // Se utiliza el organizador de diseño por omisión: BorderLayout.
    // El JScrollPane es agregado al centro.
    add(jsp, BorderLayout.CENTER);
  }
}
```

La salida del ejemplo se muestra a continuación:



JList

En Swing, la clase básica para creación de listas se llamada **JList**. Esta clase soporta la selección de uno o más elementos de la lista. Aunque la lista consiste frecuentemente en cadenas, es posible crear una lista de cualquier tipo de objetos que puedan ser desplegados. La clase **JList** es

ampliamente utilizada en Java por lo cual es altamente improbable que el lector no la haya visto anteriormente.

La clase **JList** provee varios constructores. El único utilizado aquí es:

JList(Object[ ] *elementos*)

Este constructor crea un objeto de tipo **JList** que contiene los elementos del arreglo especificado por *elementos*.

La clase **JList** está basada en dos modelos. El primero es el modelo **ListModel**. La interfaz **ListModel** define cómo se logra acceder a los datos de la lista. El segundo modelo es la interfaz **ListSelectionModel**, la cual define métodos que determinan qué elemento o elementos están seleccionados.

Aunque una **JList** puede funcionar adecuadamente por sí misma, regularmente se envolverá a la **JList** dentro de un **JScrollPane**. De esta forma, una lista larga será automáticamente desplazable, lo cual simplifica el diseño de la interfaz gráfica. Esto también facilita cambiar el número de entradas en una lista sin tener que cambiar el tamaño del componente **JList**.

Un **JList** genera un **ListSelectionEvent** cuando el usuario realiza o cambia una selección. Este evento también se genera cuando el usuario deselecciona un elemento. Estos eventos se gestionan implementando a **ListSelectionListener**. Este listener especifica sólo un método, llamado **valueChanged( )**, el cual se muestra a continuación:

void valueChanged(ListSelectionEvent *le*)

Donde, *le* es una referencia al objeto que generó el evento. Aunque **ListSelectionEvent** provee algunos métodos propios, normalmente se consulta al objeto **JList** para determinar qué ha ocurrido. Ambos, **ListSelectionEvent** y **ListSelectionListener** están contenidos en el paquete **javax.swing.event**.

Por omisión, un **JList** permite que el usuario seleccione múltiples rangos de elementos dentro de la lista, pero se puede cambiar este comportamiento llamando al método **setSelectionMode( )**, el cual está definido por **JList**. Este método se muestra a continuación:

void setSelectionMode(int *modo*)

Donde, *modo* especifica el modo de selección y debe ser uno de los siguientes valores definidos por **ListSelectionModel**:

SINGLE_SELECTION

SINGLE_INTERVAL_SELECTION

MULTIPLE_INTERVAL_SELECTION

Por omisión, la selección de múltiples intervalos, permite al usuario seleccionar múltiples rangos de elementos dentro de una lista. Con una selección de intervalos simples, el usuario puede solamente seleccionar un rango de elementos. Con la selección simple, el usuario sólo puede seleccionar un elemento. Claro está, que en los dos primeros modos también es posible seleccionar un sólo elemento. Simplemente que ellos también permiten seleccionar un rango.

Se puede obtener el índice del primer elemento seleccionado, el cual también será el índice del único elemento seleccionado cuando se utiliza el modo de selección simple, llamando al método **getSelectedIndex( )**, que se muestra a continuación:

int getSelectedIndex( )

El indexado comienza con cero. Por lo tanto, si se selecciona al primer elemento, este método devuelve 0. Si no hay elementos seleccionados, se devuelve -1.

En lugar de obtener el índice de una selección, se puede obtener el valor asociado con la selección llamando al método **getSelectedValue( )**:

Object getSelectedValue( )

Este método devuelve una referencia al primer valor seleccionado. Si ningún valor ha sido seleccionado, se devuelve **null**.

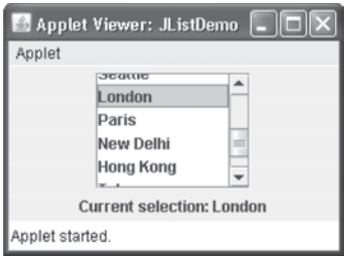
El siguiente applet muestra un ejemplo simple de **JList**, el cual contiene una lista de ciudades. Cada vez que una ciudad es seleccionada de la lista, se genera un evento **ListSelectionEvent**, el cual es gestionado por el método **valueChanged( )** definido por **ListSelectionListener**. Éste responde obteniendo el índice del elemento seleccionado y mostrando el nombre de la ciudad seleccionada en una etiqueta.

```
// Ejemplo de JList.
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;

/*
  <applet code="JListDemo" width=200 height=120>
  </applet>
*/

public class JListDemo, extends JApplet {
  JList jlst;
  JLabel jlab;
  JScrollPane jscrlp;

  // Crea un arreglo de ciudades.
  String cuidades[] = { "Nueva York", "Chicago", "Houston",
                        "Denver", "Los Ángeles", "Seattle",
                        "Londres", "París", "Nueva Delhi",
                        "Hong Kong", "Tokio", "Sydney"};

  public void init() {
    try {
      SwingUtilities.invokeAndWait(
        new Runnable() {
          public void run() {
            makeGUI();
          }
        }
      );
    } catch (Exception exc) {
      System.out.println("No se puede crear la aplicación debido a: " + exc);
    }
  }

  private void makeGUI() {

    // Cambia a FlowLayout.
    setLayout(new FlowLayout());
```

PARTE III

```
    // Crea una JList.
    jlst = new JList(ciudades);

    // Define el modo de selección de la lista a modo simple
    jlst.setSeleetionMode(ListSeleetionModel.SINGLE_SELECTION);

    // Agrega la lista a un JScrollPane.
    jscrlp = new JScrollPane(jlst);

    // Define el tamaño preferido del JScrollPane.
    jscrlp.setPreferredSize(new Dimension(120, 90));

    // Crea una etiqueta que muestra la selección.
    jlab = new JLabel("Seleccione una ciudad");

    // Agrega un listener de selección a la lista
    jlst.addListSelectionListener(new ListSelectionListener() {
      public void valueChanged(ListSelectionEvent le) {
        // Obtiene el índice del elemento modificado.
        int idx = jlst.getSelectedIndex();

        // Muestra la selección si el elemento fue seleccionado.
        if (idx != -1)
          jlab.setText("Selección Actual: " + ciudades[idx]);
        else // en caso contrario, preguntar la ciudad.
          jlab.setText("Seleccione una ciudad");

      }
    });

    // Agrega la lista y una etiqueta al contenedor principal
    add(jscrlp);
    add(jlab);
  }
}
```

La salida del ejemplo de listas se muestra a continuación:



JComboBox

Swing proporciona un componente que es una *combinación de un campo* de texto y una lista desplegable a través de la clase **JComboBox**. Un objeto JComboBox normalmente despliega una entrada, pero también cuenta con una lista desplegable que permite al usuario seleccionar una entrada diferente. También es posible crear un combo box que permita al usuario ingresar una selección dentro del campo de texto. El constructor **JComboBox** utilizado por el ejemplo se muestra a continuación:

JComboBox(Object[ ] *elementos*)

Donde, *elementos* es un arreglo que inicializa al objeto **JComboBox**. Muchos otros constructores están disponibles.

La clase **JComboBox** utiliza a la clase **ComboBoxModel**. Los objetos **JComboBox** que pueden ser modificados (esos cuyas entradas pueden ser cambiadas) utilizan a **MutableComboBoxModel**.

Adicionalmente, para pasar un arreglo de elementos para que sean mostrados en la lista, los elementos pueden agregarse dinámicamente a la lista vía el método **addItem( )**, mostrado a continuación:

void addItem(Object *obj*)

Donde, *obj* es el objeto a ser agregado al **JComboBox**. Este método debe ser utilizado sólo con **JComboBox** modificables.

La clase **JComboBox** genera un evento de acción cuando el usuario selecciona un elemento de la lista. La clase **JComboBox** también genera un evento de tipo elemento cuando el estado de selección cambia, lo cuál ocurre cuando un elemento se selecciona o deselecciona. Por ello, cambiar una selección causa que ocurran dos eventos de tipo elemento: uno para el elemento deseleccionado y otro para el elemento seleccionado. Frecuentemente, es suficiente simplemente escuchar los eventos de acción, aunque ambos tipos de eventos están disponibles para su uso.

Una forma de obtener el elemento seleccionado en la lista es llamando al método **getSelectedItem( )** del objeto **JComboBox**. Este método se muestra a continuación:

Object getSelectedItem( )

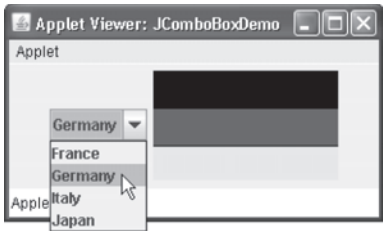
Se necesitará hacer una conversión del valor devuelto al tipo de objeto almacenado en la lista.

El siguiente ejemplo, muestra el uso de **JComboBox**. El objeto **JComboBox** contiene entradas para"Francia","Alemania","Italia"y"Japón". Cuando un país es seleccionado, una etiqueta que contiene un icono se actualiza para mostrar la bandera del país. Se puede ver lo pequeño del código requerido para utilizar este poderoso componente.

```
// Ejemplo de JComboBox.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
  <applet code="JComboBoxDemo" width=300 height=100>
  </applet>
*/

public class JComboBoxDemo extends JApplet {
  JLabel jlab;
  ImageIcon francia, alemania, italia, japon;
  JComboBox jcb;

  String banderas[] = {"Francia", "Alemania", "Italia", "Japón" };

  public void init() {
    try {
      SwingUtilities.invokeAndWait(
        new Runnable() {
          public void run() {
```

```
            makeGUI();
          }
        }
      );
    } catch (Exception exc) {
       System.out.println("No se puede crear la aplicación debido a: " + exc);
    }
  }

  private void makeGUI() {

    // Cambia a FlowLayout.
    setLayout(new FlowLayout());

    // Crea un JComboBox y lo agrega al contenedor principal
    jcb = new JComboBox(banderas);
    add(jcb);

    // Gestiona las selecciones.
    jcb.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent ae) {
       String s = (String) jcb.getSelectedItem();
       jlab.setIcon(new ImageIcon(s + ".gif"));
      }
    });

    // Crea una etiqueta y la agrega al contenedor principal
    jlab = new JLabel(new ImageIcon("Francia.gif"));
    add(jlab);
  }
}
```

La salida del ejemplo se muestra a continuación:



JTree

Un *árbol* es un componente que presenta una vista jerárquica de datos. El usuario tiene la habilidad de expandir o comprimir subárboles individuales en la vista. Los árboles son implementados en Swing por la clase **JTree**. Una muestra de sus constructores son los siguientes:

JTree(Object *obj*[ ])
JTree(Vector<?> *v*)
JTree(TreeNode *tn*)

En la primera forma, el árbol es construido a partir de los elementos en un arreglo *obj*. La segunda forma construye el árbol a partir de los elementos del vector *v*. En la tercera forma, el árbol cuyo nodo raíz está especificado por *tn* especifica al árbol.

Aunque **JTree** está empaquetado en **javax.swing**, sus clases e interfaces de soporte están empaquetadas en el paquete **javax.swing.tree**. Esto se debe a que la lista de clases e interfaces requeridas para soportar **JTree** es un poco larga.

JTree se basa en dos modelos: **TreeModel** y **TreeSelectionModel**. Un **JTree** genera una variedad de eventos, pero tres eventos están específicamente relacionados con árboles: **TreeExpansionEvent**, **TreeSelectionEvent**, y **TreeModelEvent**. Los eventos **TreeExpansionEvent** ocurren cuando un nodo es expandido o comprimido. Un **TreeSelectionEvent** se genera cuando el usuario selecciona o deselecciona un nodo dentro del árbol. Un **TreeModelEvent** es lanzado cuando los datos o estructura del árbol cambian. Los listeners para esos eventos son **TreeExpansionListener**, **TreeSelectionListener** y **TreeModelListener**, respectivamente. Las clases e interfaces para gestión de eventos relativos a árboles están empaquetadas en **javax.swing.event**.

El evento gestionado por el programa de ejemplo mostrado en esta sección es un **TreeSelectionEvent**. Para escuchar este evento, se implementa a **TreeSelectionListener**. Este listener define sólo un método, llamado **valueChanged( )**, el cual recibe al objeto **TreeSelectionEvent**. Se puede obtener la ruta al objeto seleccionado llamando al método **getPath( )**, mostrado a continuación:

```
TreePath getPath( )
```

Este método devuelve un objeto **TreePath** que describe la ruta del nodo modificado. La clase **TreePath** encapsula información referente a la ruta de un nodo particular en el árbol. Esta clase provee varios constructores y métodos. En este libro, sólo el método **toString( )** será utilizado. Este método devuelve una cadena que describe la ruta.

La interfaz **TreeNode** declara métodos que obtienen información referente a un nodo del árbol. Por ejemplo, es posible obtener una referencia al nodo padre o una enumeración de los nodos hijos. La interfaz **MutableTreeNode** extiende de **TreeNode**. Esta interfaz declara métodos que pueden insertar y remover nodos hijos o cambiar al nodo padre.

La clase **DefaultMutableTreeNode** implementa a la interfaz **MutableTreeNode**. Ésta representa a un nodo en un árbol. Uno de sus constructores se muestra aquí:

```
DefaultMutableTreeNode(Object obj)
```

Donde, *obj* es el objeto que será incluido en el nodo del árbol. El nuevo nodo del árbol no tiene un padre o hijo.

Para crear una jerarquía de nodos en el árbol, se puede utilizar el método **add( )** definido en **DefaultMutableTreeNode**. Su firma se muestra a continuación:

```
void add(MutableTreeNode hijo)
```

Donde, *hijo* es un nodo modificable del árbol que será agregado como hijo del nodo actual.

La clase **JTree** no provee ninguna capacidad de desplazamiento por sí misma. En lugar de eso, un **JTree** típicamente es colocado dentro de un **JScrollPane**. De esta forma, un árbol grande puede ser desplazado a través de un viewport pequeño.

Aquí están los pasos a seguir para usar un árbol:

1. Crear una instancia de la clase **JTree.**
2. Crear un objeto **JScrollPane** y especificar al árbol como el objeto a ser desplazado.
3. Agregar el árbol en el **JScrollPane**.
4. Agregar el **JScrollPane** en el contenedor principal.

El siguiente ejemplo muestra cómo crear un árbol y gestionar selecciones. El programa crea una instancia de **DefaultMutableTreeNode** etiquetada como "Opciones"Este es el nodo en la parte superior en la jerarquía del árbol. Adicionalmente, se crean tres nodos en el árbol, y se llama al método **add( )** para conectar esos nodos al árbol. Una referencia al nodo superior en el árbol se provee como argumento para el constructor de **JTree**. El árbol es entonces utilizado como argumento para el constructor de **JScrollPane**. Este **JScrollPane** es agregado al contenedor principal. A continuación, se crea una etiqueta y se agrega al contenedor principal. La selección del árbol se muestra en esta etiqueta. Para recibir la selección de eventos del árbol, se registra un **TreeSelectionListener** para el árbol. Dentro del método **valueChanged( )**, la ruta a la selección actual es obtenida y mostrada.

```
// Ejemplo de JTree.
import java.awt.*;
import javax.swing.event.*;
import javax.swing.*;
import javax.swing.tree.*;
/*
  <applet code="JTreeDemo" width=400 height=200>
  </applet>
*/

public class JTreeDemo extends JApplet {
  JTree arbol;
  JLabel jlab;

  public void init() {
    try {
      SwingUtilities.invokeAndWait(
        new Runnable() {
          public void run() {
            makeGUI();
          }
        }
      );
    } catch (Exception exc) {
      System.out.println("No se puede crear la aplicación debido a: " + exc);
    }
  }

  private void makeGUI() {

    // Crea el nodo superior del árbol
    DefaultMutableTreeNode top = new DefaultMutableTreeNode("Opciones");

    // Crea un subárbol "A".
    DefaultMutableTreeNode a = new DefaultMutableTreeNode("A");
    top.add(a);
    DefaultMutableTreeNode a1 = new DefaultMutableTreeNode("A1");
```

```
    a.add(a1);
    DefaultMutableTreeNode a2 = new DefaultMutableTreeNode("A2");
    a.add(a2);

    // Crea un subárbol "B".
    DefaultMutableTreeNode b = new DefaultMutableTreeNode("B");
    top.add(b);
    DefaultMutableTreeNode b1 = new DefaultMutableTreeNode("B1");
    b.add(b1);
    DefaultMutableTreeNode b2 = new DefaultMutableTreeNode("B2");
    b.add(b2);
    DefaultMutableTreeNode b3 = new DefaultMutableTreeNode("B3");
    b.add(b3);

    // Crea el árbol
    arbol = new JTree(top);

    // Agrega el árbol a un JScrollPane.
    JScrollPane jsp = new JScrollPane(arbol);

    // Agrega el JScrollPane al contenedor principal
    add(jsp);

    // Agrega la etiqueta al contenedor principal
    jlab = new JLabel();
    add(jlab, BorderLayout.SOUTH);

    // Gestiona los eventos de selección en el árbol.
    tree.addTreeSeleetionListener(new TreeSeleetionListener()
      public void valueChanged(TreeSeleetionEvent tse)
        jlab.setText("La selección es " + tse.getPath());
      }
    });
  }
}
```

La salida del ejemplo de árbol se muestra aquí:

La cadena presentada en el campo de texto representa la ruta desde el nodo superior del árbol hasta el nodo seleccionado.

JTable

JTable es un componente que muestra filas y columnas de datos. Se puede arrastrar el cursor sobre los límites de una columna para cambiar el tamaño de la columna. También se puede arrastrar a una columna a una nueva posición. Además dependiendo de su configuración, es posible seleccionar una fila, una columna o una celda dentro de la tabla, y cambiar sus datos. **JTable** es un componente sofisticado que ofrece más opciones y características de las que pueden ser discutidas aquí. Este es quizá el componente más complejo de Swing. Sin embargo, en su configuración por omisión, **JTable** ofrece una funcionalidad sustancial que es fácil de utilizar, especialmente si simplemente se desea utilizar la tabla para presentar datos en una forma tabular. La breve descripción presentada aquí nos dará una visión general de este poderoso componente.

Tal como ocurre con **JTree**, **JTable** tiene muchas clases e interfaces asociadas con él. Estas clases están empaquetadas en **javax.swing.table**.

JTable es un componente simple en esencia. Es un componente que consiste de una o más columnas de información. En la parte superior de cada columna existe un encabezado. Además de describir los datos de la columna, el encabezado proporciona el mecanismo mediante el cual el usuario puede cambiar el tamaño de la columna o cambiar la ubicación de la columna dentro de la tabla. **JTable** no provee ninguna capacidad de desplazamiento por sí misma. En su lugar, normalmente se envolverá a **JTable** dentro de un **JScrollPane**.

JTable tiene varios constructores. El único utilizado aquí es:

JTable (Object *datos*[ ][ ], Object *encabezadosColumnas*[ ])

Donde, *datos* es un arreglo bidimensional que contiene la información a ser presentada y *encabezadosColumnas* es un arreglo unidimensional con los encabezados de las columnas.

JTable se basa en tres modelos. El primero es el modelo de tabla, el cual está definido por la interfaz **TableModel**. Este modelo define lo necesario para desplegar los datos en un formato de dos dimensiones. El segundo modelo es el modelo columnas, el cual está representado por **TableColumnModel**. **JTable** está definido en términos de columnas, y su **TableColumnModel** especifica las características de una columna. Estos dos modelos están empaquetados en **javax.swing.table**. El tercer modelo determina cómo los elementos son seleccionados, y está especificado por **ListSelectionModel**, el cual fue descrito cuando describimos al componente **JList**.

Un **JTable** puede generar muchos eventos diferentes. Los dos eventos más importantes para la operación de una tabla son **ListSelectionEvent** y **TableModelEvent**. Un evento **ListSelectionEvent** se genera cuando el usuario selecciona algo en la tabla. Por omisión **JTable** permite seleccionar una o más líneas completas, pero se puede cambiar ese comportamiento para permitir que una o más columnas o una o más celdas individuales sean seleccionadas. Un evento **TableModelEvent** se genera cuando los datos de la tabla cambian de alguna forma. Gestionar este evento requiere un poco más de trabajo del que se requiere para gestionar los eventos generados por los componentes descritos anteriormente y está fuera del alcance de este libro. Sin embargo, si simplemente se quiere utilizar un **JTable** para mostrar información (como en el siguiente ejemplo), entonces no necesita gestionar ningún evento.

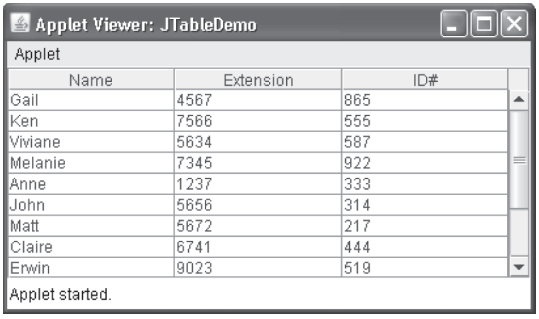
A continuación se muestran los pasos a seguir para crear un **JTable** que puede ser utilizado para mostrar información:

1. Crear una instancia de **JTable**.
2. Crear un objeto **JScrollPane**, especificando la tabla como objeto a ser desplazado.
3. Agregar la tabla al **JScrollPane**.
4. Agregar el **JScrollPane** al contenedor principal.

El siguiente ejemplo ilustra cómo crear y utilizar una tabla simple. Un arreglo unidimensional de cadenas llamado **encabezadosColumna** contiene los encabezados de las columnas. Un arreglo bidimensional de cadenas llamado dato contiene las celdas de la tabla. Como se puede ver cada elemento en el arreglo es un arreglo de tres cadenas. Esos arreglos se pasan al constructor del **JTable**. La tabla se agrega a un **JScrollPane**, y luego el **JScrollPane** se agrega al contenedor principal. Los datos del arreglo **dato** son mostrados en la tabla. Por omisión la configuración de la tabla permite que el contenido de una celda pueda ser editado. Los cambios afectan al arreglo subyacente, **dato** en este caso.

```
// Ejemplo de JTable.
import java.awt.*;
import javax.swing.*;
/*
  <applet code="JTableDemo" width=400 height=200>
  </applet>
*/

public class JTableDemo extends JApplet {

  public void init() {
    try {
      SwingUtilities.invokeAndWait(
        new Runnable() {
          public void run()
            makeGUI();
          }
        }
      );
    } catch (Exception exc) {
      System.out.println("No se puede crear la aplicación debido a: " + exc);
    }
  }

  private void makeGUI() {

    // Inicializa los encabezados de las columnas
    String [] encabezadosColumna = { "Nombre", "Extensión", "ID" };

    // inicializa el arreglo dato.
    Obj ect [][J dato = {
      {"Javier" , "4567", "865"},
      {"Helen", "7566", "555"},
      {"Rosana", "5634", "587"},
      {"Adriana", "7345", "922"},
      {"Ana", "1237", "333"},
      {"José", "5656", "314"},
      {"Mario", "5672", "217"},
      {"Clara", "6741", "444"},
```

```
      {"Ernesto", "9023", "519"},
      {"Ken", "1134", "532"},
      {"Teresa", "5689", "112"},
      {"Elizabeth", "9030", "133"},
      {"Arturo", "6751", "145"}
    };

    //Crea la tabla
    JTable table = new JTable(dato, encabezadosColumna);

    // Agrega la tabla al JScrollPane
    JScrollPane jsp = new JScrollPane (table);

    //Agrega el JScrollPane al contenedor principal
    add(jsp);
  }
}
```



La salida del ejemplo se muestra a continuación:

Otras características para explorar de Swing

Swing define una gran cantidad de componentes y herramientas. Swing cuenta con una gran cantidad de características que seguramente el lector deseará explorar por su cuenta. Por ejemplo, Swing permite la creación de barras de herramientas, herramientas de ayuda, y barras de progreso. También provee un completo subsistema de menús y la posibilidad de configurar la apariencia de la aplicación. Con Swing es posible definir modelos propios para los componentes y cambiar la forma en que las celdas son editadas y dibujadas cuando se trabaja con tablas y árboles. La mejor forma de familiarizarse con las capacidades de Swing es experimentar con él.

31 CAPÍTULO

Servlets

Este capítulo presenta una descripción general de los *servlets*. Los servlets son pequeños programas que se ejecutan del lado del servidor en una conexión Web. Así como los applets dinámicamente extienden la funcionalidad de un navegador Web, los servlets dinámicamente extienden la funcionalidad de un servidor Web. El tema de servlets es amplio, y está más allá del alcance de este capítulo cubrirlo completamente. En lugar de ello, nos enfocaremos en los conceptos, interfaces y clases clave, así como en diversos ejemplos.

Introducción

Para comprender las ventajas proporcionadas por los servlets, es necesario comprender cómo los navegadores y servidores Web cooperan para proporcionar información al usuario. Considere una petición por una página Web estática. El usuario escribe un URL en el navegador. El navegador genera una petición HTTP al servidor Web correspondiente. El servidor Web proyecta la petición a un archivo específico. Ese archivo es devuelto en una respuesta HTTP al navegador. El encabezado HTTP en la respuesta, indica el tipo del contenido. Para especificar el tipo de contenido se utilizan tipos MIME (Multipurpose Internet Mail Extensions). Por ejemplo, el texto convencional con caracteres ASCII tiene como tipo MIME text/plain. El código HTML de una página Web tiene como tipo MIME text/html.

Ahora consideremos el contenido dinámico. Asumamos que una tienda en línea utiliza una base de datos para almacenar información acerca de su negocio. Esto incluye elementos en venta, precios, disponibilidad, órdenes de compra, etc. La tienda desea que esta información pueda ser consultada por sus clientes a través de páginas Web. El contenido de dichas páginas debe ser generado dinámicamente para reflejar la información más reciente de la base de datos.

En los inicios del Web, un servidor podía crear páginas de manera dinámica creando procesos separados para gestionar la petición de cada cliente. El proceso podría abrir conexiones con una o más bases de datos para obtener la información necesaria. Luego comunicarse con el servidor Web a través de una interfaz conocida como CGI (Common Gateway Interface). CGI permite a un proceso independiente leer datos desde una petición HTTP y escribir datos en la respuesta HTTP. Una variedad de lenguajes fueron utilizados para construir programas con CGI. Estos lenguajes incluyen C, C++ y Perl.

Sin embargo, CGI padece de serios problemas de rendimiento. Es muy caro en términos de uso de recursos (procesador y memoria) ya que debe crear procesos separados para cada petición. De igual forma resultaba caro abrir y cerrar conexiones a las bases de datos para cada petición de un cliente.

Además los programas hechos con CGI no eran independientes de la plataforma. Por lo anterior, otras técnicas fueron desarrolladas. Entre ellas los servlets.

Los servlets ofrecen diversas ventajas en comparación con CGI. Primero, el rendimiento es significativamente mejor. Los servlets se ejecutan en el espacio de direcciones del servidor Web. No es necesario crear un proceso separado para gestionar cada petición del cliente. Segundo, los servlets son independientes de la plataforma debido a que están escritos en Java. Tercero, el administrador de seguridad de Java en el servidor impone un conjunto de restricciones para proteger los recursos del servidor. Finalmente, toda la funcionalidad de las bibliotecas de Java está disponible para los servlets. Estos pueden comunicarse con applets, bases de datos, y otros programas a través de sockets y el mecanismo RMI visto antes.

El ciclo de vida de un servlet

Tres métodos son fundamentales en el ciclo de vida de un servlet. Estos métodos son **init( )**, **service( )** y **destroy( )**. Estos se implementan por todos los servlets y son invocados en momentos específicos por el servidor. Consideremos un escenario tradicional para entender cuándo son llamados estos métodos.

Primero, asumamos que un usuario escribe un URL en un navegador Web. El navegador genera una petición HTTP para este URL. Esta petición se envía al servidor apropiado.

Segundo, la petición HTTP es recibida por el servidor Web. El servidor envía la petición a un servlet específico. El servlet es dinámicamente ejecutado en el espacio de direcciones del servidor.

Tercero, el servidor invoca al método **init( )** del servlet. Este método es invocado sólo cuando el servlet es cargado en memoria por primera vez. Es posible enviar parámetros de inicialización al servlet para que éste sea configurado.

Cuarto, el servidor invoca al método **service( )** del servlet. Este método se llama para procesar la petición HTTP. Veremos que es posible para el servlet leer datos que se le envían en la petición HTTP. Además el servlet puede elaborar una respuesta HTTP para el cliente.

El servlet permanece disponible en el espacio de direcciones del servidor listo para procesar más peticiones HTTP que lleguen desde el cliente. El método **service( )** es llamado para cada petición HTTP.

Finalmente, el servidor puede decidir quitar de su memoria al servlet. Los algoritmos por los cuales se realiza esta determinación son específicos de cada servidor. El servidor llama al método **destroy( )** para liberar los recursos utilizados por el servlet. Los datos importantes deben ser guardados en un almacenamiento persistente. La memoria asignada al servlet y sus objetos son recolectados por el recolector de basura de Java.

Uso de tomcat para el desarrollo de servlet

Para crear servlets necesitamos tener acceso a un ambiente de desarrollo de servlets. El ambiente utilizado por los ejemplos de este capítulo es Tomcat. Tomcat es un producto de código abierto sustentado por el grupo Jakarta Project de Apache Software Foundation. Tomcat contiene las bibliotecas de clases, documentación y ambiente de ejecución que necesitamos para crear y probar servlets. Al momento de escribir este libro, la versión actual de Tomcat es la 5.5.17; esta versión soporta la especificación de servlet 2.4. Tomcat puede obtenerse en la dirección Web jakarta.apache.org.

Los ejemplos en este capítulo asumen un ambiente de ejecución Windows. La ubicación de Tomcat por omisión es

```
C:\Program Files\Apache Software Foundation\Tomcat 5.5\
```

Ésta es la ubicación que asumen los ejemplos en este libro. Si se instala Tomcat en una ubicación diferente, será necesario realizar los cambios pertinentes en los ejemplos. Será necesario establecer el valor de la variable de ambiente **JAVA_HOME** al valor del directorio raíz donde está instalado el ambiente de desarrollo de Java.

Para iniciar Tomcat, se selecciona "Configure Tomcat" en "Start | Programs menu" y luego se presiona "Start" en el cuadro de propiedades de Tomcat.

Cuando ya no necesitemos trabajar con servlets, podemos detener a Tomcat presionando "Stop" en el cuadro de propiedades de Tomcat.

El directorio

```
C:\Program Files\Apache Software Foundation\Tomcat 5.5\common\lib\
```

Contiene el archivo **servlet-api.jar**. Este archivo JAR contiene las clases e interfaces que se necesitan para construir servlets. Para que este archivo sea accesible, se actualiza la variable de ambiente **CLASSPATH** para incluirle

```
C:\Program Files\Apache Software Foundation\Tomcat 5.5\common\lib\servlet-api.jar
```

Alternativamente, es posible especificar el archivo cuando se compila el servlet. Por ejemplo, el siguiente comando compila al servlet del ejemplo anterior:

```
javac HelloServlet.java –classpath "C:\Program Files\Apache Software Founda-
tion\Tomcat 5.5\common\lib\servlet-api.jar"
```

Una vez que se ha compilado el servlet, se le debe indicar a Tomcat cómo localizarlo. Esto significa que se debe colocar al servlet en un directorio dentro del directorio **webapps** de Tomcat y colocar el nombre del servlet en un archivo **web.xml**. Para mantener las cosas simples, el ejemplo en este capítulo utiliza al directorio y archivo **web.xml** que Tomcat proporciona para sus propios servlets de ejemplo. Aquí está el procedimiento que se debe seguir.

Primero, copie el archivo *.class del servlet en el siguiente directorio:

```
C:\Program Files\Apache Software Foundation\Tomcat 5.5\webapps
  \servlets-examples\WEB-INF\classes
```

Luego, se añade el nombre del servlet y se mapea al archivo **web.xml** en el siguiente directorio:

```
C:\Program Files\Apache Software Foundation\Tomcat 5.5\webapps
  \servlets-examples\WEB-INF
```

Por ejemplo, para la clase llamada **HelloServlet**, se deben añadir las siguientes líneas en la sección de definición de servlets:

```
<servlet>
  <servlet-name>HelloServlet</servlet-name>
  <servlet-class>HelloServlet</servlet-class>
</servlet>
```

A continuación, se añaden las siguientes líneas a la sección que define el mapeo de servlets.

```
<servlet-mapping>
<servlet-name>HelloServlet</servlet-name>
```

```
<url-pattern>/servlet/HelloServlet</url-pattern>
</servlet-mapping>
```

El mismo procedimiento general se debe seguir para todos los ejemplos.

Un servlet sencillo

Para familiarizarnos con los conceptos clave de los servlets, construyamos y probemos un servlet simple. Los pasos básicos son los siguientes:

1. Crear y compilar el archivo fuente del servlet. Luego copiamos el archivo class del servlet al directorio apropiado, y se añade el nombre del servlet y su mapeo en el archivo **web.xml**.
2. Iniciar Tomcat
3. Iniciar un navegador Web y enviar una petición al servlet.

Examinemos cada uno de esos tres pasos con detalle.

Crear y compilar el código fuente de un servlet

Para comenzar, creamos un archivo llamado **HelloServlet.java** que contiene el siguiente programa:

```
import java.io.*;
import javax.servlet.*;

public class HelloServlet extends GenericServlet {

  public void service(ServletRequest request, ServletResponse response)
  throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter pw = response.getWriter();
    pw.print("<B>Hola");
    pw.close();
  }
}
```

Analicemos el programa anterior. Primero, observe que se importa el paquete **javax.servlet**. Este paquete contiene las clases e interfaces requeridas para construir servlets. Aprenderemos más de estos paquetes más adelante en este capítulo. Luego, el programa define la clase **HelloServlet** como una subclase de **GenericServlet**. La clase **GenericServlet** proporciona la funcionalidad necesaria para facilitar la creación de un servlet. Por ejemplo, proporciona los métodos **init( )** y **destroy( )**, los cuales pueden ser utilizados tal como están. Sólo es necesario que el programador escriba el método **service( )**.

Dentro de la clase **HelloServlet**, el método **service( )** (el cual es heredado de **GenericServlet**) se sobrescribe. Este método gestiona las peticiones del cliente. Note que el primer argumento es un objeto **ServletRequest**. Este objeto le permite al servlet leer datos que son proporcionados por la petición del cliente. El segundo argumento es un objeto **ServletResponse**. Este objeto le permite al servlet formular una respuesta al cliente.

La llamada al método **setContentType( )** establece al tipo MIME para la respuesta HTTP. En este programa, el tipo MIME es text/html. Esto indica al navegador que debe interpretar el contenido como código fuente HTML.

Luego, el método **getWriter( )** obtiene una referencia a **PrintWriter**. Cualquier cosa escrita en este flujo es enviada al cliente como parte de la respuesta HTTP. El método **println( )** es utilizado para escribir código HTML en la respuesta HTTP.

Ahora compilemos este programa y coloquemos el archivo **HelloServlet.class** en el directorio de Tomcat indicado en la sección anterior. Y agreguemos a **HelloServlet** en el archivo **web.xml**, como se describió antes.

Arrancando el servidor web Tomcat

Tomcat se arranca como se explicó antes. Tomcat debe estar corriendo antes de que intentemos ejecutar un servlet.

Acceso al servlet con un navegador

Finalmente abrimos un navegador web y escribimos la dirección URL mostrada abajo:

```
http://localhost:8080/servlets-examples/servlet/HelloServlet
```

Alternativamente, se puede escribir el URL así:

```
http://127.0.0.1:8080/servlets-examples/servlet/HelloServlet
```

Esto se puede realizar debido a que 127.0.0.1 está definida como la dirección IP equivalente a localhost.

La salida del servlet puede ser observada en el navegador. La salida es el texto "**Hola**" en negritas.

El servlet API

Son dos los paquetes que contienen las clases e interfaces que se requieren para construir servlets. Estos son **javax.servlet** y **javax.servlet.http**. Estos paquetes constituyen el API de desarrollo de servlets. Recuerde que estos paquetes no son parte de los paquetes en el núcleo de Java por ello no están incluidos en Java SE 6. Sin embargo, son extensiones, estándar proporcionadas por Tomcat.

El API para el desarrollo de servlets continúa en proceso de desarrollo y mejora. La especificación actual de servlets es la versión 2.4, y es la utilizada en este libro. Sin embargo, debido a la rapidez con que ocurren los cambios en el mundo de Java, se recomienda al lector revisar por nuevas actualizaciones y cambios. Este capítulo revisa el núcleo del API para desarrollo de servlets.

El paquete javax.servlet

El paquete **javax.servlet** contiene las clases e interfaces que establecen la estructura fundamental para la operación de los servlet. La siguiente tabla es un resumen de las interfaces principales proporcionadas por este paquete. La más significativa es **Servlet**. Todos los servlets deben implementar esta interfaz o extender de una clase que implemente esta interfaz. Las interfaces **ServletRequest** y **ServletResponse** también son muy importantes.

Interfaz	Descripción
Servlet	Declara los métodos del ciclo de vida de un servlet.
ServletConfig	Permite al servlet recibir parámetros de inicialización.
ServletContext	Permite al servlet llevar una bitácora con información sobre eventos y accesos a su ambiente.
ServletRequest	Se utiliza para leer datos de la petición del cliente.
ServletResponse	Se utiliza para escribir datos en la respuesta que se enviará al cliente.

La siguiente tabla es un resumen de las clases fundamentales disponibles en el paquete **javax.servlet**:

Clase	Descripción
GenericServlet	Implementa las interfaces **Servlet** y **ServletConfig**.
ServletInputStream	Proporciona un flujo de entrada para leer la petición del cliente.
ServletOutputStream	Proporciona un flujo de salida para escribir la respuesta al cliente.
ServletException	Indica la ocurrencia de un error en el servlet.
UnavailableException	Indica que un servlet no está disponible.

Examinemos estas clases e interfaces con detalle.

La interfaz Servlet

Todos los servlets deben implementar la interfaz **Servlet**. Esta interfaz declara los métodos **init( )**, **service( )** y **destroy( )**, los cuales serán llamados por el servidor durante el ciclo de vida del servlet. Además se proporciona un método que le permite al servlet obtener parámetros de inicialización. Los métodos definidos por la clase **Servlet** se muestran en la Tabla 31-1.

Los métodos **init( )**, **service( )** y **destroy( )** son los métodos del ciclo de vida del servlet. Estos métodos son invocados por el servidor. El método **getServletConfig( )** es invocado por el servlet para obtener los parámetros de inicialización. El programador puede además sobrescribir al método **getServletInfo( )** para que devuelva una cadena con información útil (por ejemplo, nombre del autor, versión y fecha). Este método también es invocado por el servidor.

La interfaz ServletConfig

La interfaz **ServletConfig** le permite al servlet obtener sus datos de configuración al ser cargado. Los métodos declarados por esta interfaz se resumen a continuación:

Método	Descripción
ServletContext getServletContext( )	Devuelve el contexto del servlet invocante.
String getInitParameter(String *param*)	Devuelve el valor del parámetro de inicialización llamado *param*.
Enumeration getInitParameterNames( )	Devuelve una enumeración con todos los nombres de sus parámetros de inicialización.
String getServletName()	Devuelve el nombre del servlet invocante.

La interfaz ServletContext

La interfaz **ServletContext** le permite al servlet obtener información sobre su ambiente. Muchos de los métodos de la interfaz se muestran en la Tabla 31-2.

Método	Descripción
void destroy( )	Este método es invocado cuando se da de baja al servlet.
ServletConfig getServletConfig( )	Devuelve un objeto **ServletConfig** que contiene los parámetros de inicialización.
String getServletInfo( )	Devuelve una cadena que describe al servlet.
void init(ServletConfig *sc*) throws ServletException	Este método es invocado cuando se inicializa al servlet. Los parámetros de inicialización del servlet se obtienen de *sc*. Se genera una excepción de tipo **UnavailableException** si el servlet no puede ser inicializado.
void service(ServletRequest *req*, ServletResponse *res*) throws ServletException, IOException	Este método es invocado para procesar una petición del cliente. La petición del cliente puede ser leída del objeto *req*. La respuesta para el cliente puede ser escrita en el objeto *res*. Una excepción se genera si ocurre un problema con el servlet o con una operación de E/S.

TABLA 31-1 Los métodos definidos por **Servlet**

La interfaz ServletRequest

La interfaz **ServletRequest** le permite al servlet obtener información sobre la petición del cliente. Los métodos de la interfaz se resumen en la Tabla 31-3.

La interfaz ServletResponse

La interfaz **ServletResponse** le permite al servlet formular una respuesta para el cliente. Los métodos de la interfaz se resumen en la Tabla 31-4.

Método	Descripción
Object getAttribute(String *attr*)	Devuelve el valor del atributo llamado *attr* en el servidor.
String getMimeType(String *arch*)	Devuelve el tipo MIME de un *archivo*.
String getRealPath(String *vdir*)	Devuelve el directorio real que corresponde al directorio *vdir*.
String getServerInfo( )	Devuelve información sobre el servidor.
void log(String *s*)	Escribe *s* en la bitácora del servlet.
void log(String *s*, Throwable *e*)	Escribe *s* y el trazo de la pila correspondiente a *e* en la bitácora del servlet.
void setAttribute(String *attr*, Object *val*)	Escribe en el atributo especificado por *attr* el valor pasado en *val*.

TABLA 31-2 Métodos definidos por **ServletContext**

Método	Descripción
Object getAttribute(String *attr*)	Devuelve el valor del atributo llamado *attr*.
String getCharacterEncoding( )	Devuelve la codificación de los caracteres de la petición del cliente.
int getContentLength( )	Devuelve el tamaño de la petición del usuario. Devuelve -1 si el tamaño no se ha podido obtener.
String getContentType( )	Devuelve el tipo de la petición. Devuelve *null* si el tipo no puede ser determinando.
ServletInputStream getInputStream( ) throws IOException	Devuelve un objeto **ServletInputStream** que puede ser utilizado para leer datos binarios de la petición del usuario. Se genera una excepción de tipo **IllegalStateException** si el método **getReader()** ya ha sido invocado para leer de esta petición.
String getParameter(String *pname*)	Devuelve el valor del parámetro llamado *pname*.
Enumeration getParameterNames( )	Devuelve una enumeración con los nombres de los parámetros presentes en la petición del usuario.
String[ ] getParameterValues (String *nom*)	Devuelve un arreglo que contiene los valores asociados con el parámetro especificado por *nom*.
String getProtocol( )	Devuelve una descripción del protocolo.
BufferedReader getReader( ) throws IOException	Devuelve un objeto BufferedReader que puede ser utilizado para leer texto de la petición del usuario. Se genera una excepción de tipo **IllegalStateException** si el método **getInputStream()** ya ha sido invocado para la petición del usuario.
String getRemoteAddr( )	Devuelve la cadena equivalente a la dirección IP del cliente.
String getRemoteHost( )	Devuelve la cadena equivalente al nombre de la computadora del usuario.
String getScheme( )	Devuelve el esquema de transmisión del URL utilizado en la petición del usuario (por ejemplo, “http”, “ftp”).
String getServerName( )	Devuelve el nombre del servidor.
int getServerPort( )	Devuelve el número del puerto.

TABLA 31-3 Métodos definidos por **ServletRequest**

La clase GenericServlet

La clase **GenericServlet** proporciona la implementación de los métodos fundamentales del ciclo de vida de un servlet. La clase **GenericServlet** implementa las interfaces **Servlet** y **ServletConfig**. Además, está disponible un método para agregar cadenas a la bitácora del servidor. Las firmas de estos métodos se muestran a continuación:

void log(String *s*)
void log(String *s*, Throwable *e*)

Donde, *s* es la cadena a ser agregada en la bitácora, y *e* es una excepción que ha ocurrido.

Método	Descripción
String getCharacterEncoding()	Devuelve la codificación de los caracteres para la respuesta.
ServletOutputStream( ) throws IOException	Devuelve un objeto **ServletOutputStream** que puede ser utilizado para escribir datos binarios en la respuesta. Una excepción de tipo **IllegalStateException** se genera si el método **getWriter( )** ya ha sido invocado para la respuesta actual.
PrintWriter getWriter( ) throws IOException	Devuelve un objeto **PrintWriter** que puede ser utilizado para escribir datos en la respuesta. Una excepción de tipo **IllegalStateException** se genera si el método **getOutputStream( )** ya ha sido invocado para la salida actual.
void setContentLength(int *tam*)	Establece como tamaño de la respuesta al valor *tam*.
void setContentType(String *tipo*)	Establece como tipo para la respuesta al valor *tipo*.

TABLA 31-4 Métodos definidos por **ServletResponse**

La clase ServletInputStream

La clase **ServletInputStream** hereda de **InputStream**. Es implementada por el contenedor de servlets y proporciona un flujo de entrada que el programador del servlet puede utilizar para leer datos de la petición del cliente. La clase define al constructor por omisión. Adicionalmente, proporciona un método para leer bytes del flujo. El método es:

int readLine(byte[] *buf*, int *desp*, int *tam*) throws IOException

Aquí, *buf* es el arreglo en el cual una *cantidad* de bytes igual a *tam* son colocados a partir de la posición *desp*. El método devuelve el número actual de bytes leídos o –1 si el final del flujo es alcanzado.

La clase ServletOutputStream

La clase **ServletOutputStream** hereda de **OutputStream**. Es implementada por el contenedor de servlets y proporciona un flujo de salida que el programador del servlet puede utilizar para escribir datos en la respuesta para el cliente. La clase define al constructor por omisión. Adicionalmente, define los métodos **print( )** y **println( )** los cuales escriben información en el flujo.

La clase ServletException

El paquete **javax.servlet** define dos excepciones. La primera es **ServletException**, la cual indica que ha ocurrido un problema con el servlet. La segunda es **UnavailableException**, la cual hereda de **ServletException**. Ésta indica que el servlet no está disponible.

Leyendo parámetros de un servlet

La interfaz **ServletRequest** incluye métodos que permiten leer los nombres y valores de los parámetros que están incluidos en la petición del cliente. Para mostrar el uso de estos parámetros desarrollaremos un servlet. El ejemplo contiene dos archivos. Una página Web definida en el archivo **PostParameters.htm** y el servlet definido en el archivo **PostParametersServlet.java.**

El código fuente HTML de **PostParameters.htm** se muestra abajo. Este código define una tabla que contiene dos etiquetas y dos campos de texto. Una de las etiquetas es Empleado y la otra Teléfono. También tiene un botón de envío. Observe que el parámetro de nombre action en la etiqueta FORM especifica un URL. El URL identifica al servlet que procesara la petición.

```
<html>
<body>
<center>
<form name = "Form1"
  method = "post"
  action = "http://localhost:8080/servlets-examples/
            servlet/PostParametersServlet">
<table>
<tr>
  <td><B>Empleado</td>
  <td><input type=textbox name="e" size="25" value=""></td>
</tr>
<tr>
  <td><B>Tel&eacute;fono</td>
  <td><input type=textbox name="p" size="25" value=""></td>
</tr>
</table>
<input type=submit value="Submit">
</body>
</html>
```

El código fuente del servlet **PostParametersServlet.java** se muestra a continuación. El método **service( )** se sobrescribe para procesar las peticiones del cliente. El método **getParameterNames( )** devuelve una enumeración con los nombres de los parámetros. Los parámetros se procesan en un ciclo. Puede verse que los nombres de los parámetros y sus valores son enviados al cliente. El valor de cada parámetro se obtiene con el método **getParameter( ).**

```
import java.io.*;
import java.util.*;
import javax.servlet.*;

public class PostParametersServlet extends GenericServlet {

  public void service(ServletRequest request, ServletResponse response)
  throws ServletException, IOException {

    // obtener el flujo de salida
    PrintWriter pw = response.getWriter();

    // obtener la enumeración de parámetros
    Enumeration e = request.getParameterNames();

    // visualizar nombres y valores de los parámetros
    while (e.hasMoreElements()) {
      String pname = (String) e.nextElement();
      pw.print(pname + " = ");
      String pvalue = (String) request.getParameter(pname);
      pw.println(pvalue);
    }
    pw.close();
  }
}
```

Ahora compilemos el servlet. Luego, lo copiamos en el directorio adecuado, y actualizamos el archivo **web.xml**, como se describió anteriormente. Luego, se ejecutan los siguientes pasos para probar el ejemplo:

1. Se inicia Tomcat (si no está corriendo ya).
2. Se despliega la página Web en el navegador.
3. Escriba el nombre de un empleado y el número telefónico en los campos de texto.
4. Presione el botón enviar.

Después de seguir los pasos anteriores, el navegador desplegará una respuesta generada dinámicamente por el servlet.

El paquete javax.servlet.http

El paquete **javax.servlet.http** contiene diversas clases e interfaces utilizadas comúnmente en el desarrollo de servlets. La funcionalidad de estas clases e interfaces simplifica el proceso de construir servlets que funcionan con peticiones y respuestas bajo el protocolo HTTP.

La siguiente tabla resume las interfaces más importantes proporcionadas por el paquete **javax.servlet.http**.

Interfaz	Descripción
HttpServletRequest	Permite al servlet leer datos de una petición HTTP.
HttpServletResponse	Permite al servlet escribir datos a una respuesta HTTP.
HttpSession	Permite leer y escribir datos en una sesión.
HttpSessionBindingListener	Informa a un objeto que ha sido enlazado a una sesión o desenlazado de una sesión.

La siguiente tabla resume las clases más importantes proporcionadas por el paquete **javax.servlet.http**. La más importante de estas clases es **HttpServlet**. Los programadores de servlets comúnmente heredan sus clases de ésta para procesar peticiones HTTP.

Clase	Descripción
Cookie	Permite almacenar información del estado del servlet en la máquina del cliente.
HttpServlet	Proporciona métodos para gestionar peticiones y respuestas HTTP.
HttpSessionEvent	Encapsula un evento de cambio en la sesión.
HttpSessionBindingEvent	Indica cuando un listener ha sido ligado o desligado de un valor de sesión, o que un atributo de la sesión a cambiado.

La interfaz HttpServletRequest

La interfaz **HttpServletRequest** le permite a un servlet obtener información sobre la petición del cliente. Varios métodos de esta interfaz se muestran en la Tabla 31-5.

La interfaz HttpServletResponse

La interfaz **HttpServletResponse** le permite a un servlet formular una respuesta HTTP para el cliente. Define diversas constantes. Estas constantes corresponden a los diferentes códigos

de estado que pueden ser asignados a una respuesta HTTP. Por ejemplo, **SC_OK** indica que la petición HTTP tuvo éxito, y **SC_NOT_FOUND** indica que el recurso solicitado no está disponible. Varios métodos de esta interfaz se resumen en la Tabla 31-6.

La interfaz HttpSession

La interfaz **HttpSession** le permite al servlet leer y escribir información asociada con una sesión HTTP. Varios de los métodos de HttpSession se resumen en la Tabla 31-7. Todos esos métodos generan una excepción de tipo **IllegalStateException** si la sesión ha sido previamente invalidada.

Método	Descripción
String getAuthType( )	Devuelve el esquema de autentificación.
Cookie[ ] getCookies( )	Devuelve un arreglo con las cookies en la petición del cliente
long getDateHeader(String *campo*)	Devuelve el valor de la fecha en el campo de encabezado llamado *campo*.
String getHeader(String *campo*)	Devuelve el valor del campo de encabezado llamado *campo*.
Enumeration getHeaderNames( )	Devuelve una enumeración con los nombres de los encabezados.
int getIntHeader(String *campo*)	Devuelve el valor de tipo **int** equivalente al campo de encabezado llamado *campo*.
String getMethod( )	Devuelve el método HTTP utilizado en la petición del cliente.
String getPathInfo( )	Devuelve cualquier información localizada en el URL después de la ruta del servlet y antes de los datos de query-string.
String getPathTranslated( )	Devuelve cualquier información localizada en el URL después de la ruta del servlet y antes de los datos de query-string, después de traducida en la ruta real del servlet.
String getQueryString( )	Devuelve el query-string del URL.
String getRemoteUser( )	Devuelve el nombre del usuario quien realizó la petición.
String getRequestedSessionId( )	Devuelve el ID de la sesión.
String getRequestURI( )	Devuelve el URI.
StringBuffer getRequestURL( )	Devuelve el URL.
String getServletPath( )	Devuelve la parte del URL que identifica al servlet.
HttpSession getSession( )	Devuelve la sesión para esta petición. Si una sesión no existe, se crea una y luego es devuelta.
HttpSession getSession(boolean *nuevo*)	Si *nuevo* tiene el valor **true** y no existe una sesión, crea y regresa una sesión para esta petición del cliente. En caso contrario, devuelve la sesión actual para la petición.
boolean isRequestedSessionIdFromCookie( )	Devuelve **true** si una cookie contiene el ID de una sesión. En caso contrario, devuelve **false**.
boolean isRequestedSessionIdFromURL( )	Devuelve **true** si el URL contiene el ID de una sesión. En caso contrario, devuelve **false**.
boolean isRequestedSessionIdValid( )	Devuelve **true** si el ID de sesión solicitado es válido en el contexto de sesión actual.

TABLA 31-5 Métodos definidos por **H**

Método	Descripción
void addCookie(Cookie *cookie*)	Añade una *cookie* a la respuesta HTTP.
Boolean containsHeader(String *campo*)	Devuelve **true** si la respuesta HTTP contiene un campo llamado *campo*.
String encodeURL(String *url*)	Determina si el ID de la sesión debe ser codificado en el URL identificado como *url*. Si debe serlo, devuelve la versión modificada del *url*. En caso contrario, devuelve *url*. Todos los URL generados por un servlet deben ser procesados por este método.
String encodeRedirectURL(String *url*)	Determina si el ID de la sesión debe ser codificado en el URL identificado como *url*. Si debe serlo, devuelve la versión modificada del *url*. En caso contrario, devuelve *url*. Todos los URL pasados a **sendRedirect()** deben ser procesados por este método.
void sendError(int *c*) throws IOException	Envía el código de error *c* al cliente.
void sendError(int *c*, String *s*) throws IOException	Envía el código de error *c* y el mensaje *s* al cliente.
void sendRedirect(String *url*) throws IOException	Redirecciona al cliente a la dirección URL.
void setDateHeader(String *campo*, long *mseg*)	Añade *campo* al encabezado con el valor de fecha igual a *mseg* (milisegundos desde la media noche del 1 de enero de 1970 GMT).
void setHeader(String *campo*, String *valor*)	Añade *campo* al encabezado con el valor definido en *valor*.
void setIntHeader(String *campo*, int *valor*)	Añade *campo* al encabezado con el *valor* definido en valor.
void setStatus(int *c*)	Establece el código de estado para esta respuesta con el valor *c*.

TABLA 31-6 Métodos definidos por **HttpServletResponse**

La interfaz HttpSessionBindingListener

La Interfaz **HttpSessionBindingListener** es implementada por los objetos que necesitan ser notificados cuando son vinculados o desvinculados de una sesión HTTP. Los métodos que son invocados cuando un objeto es vinculado o desvinculado son:

void valueBound(HttpSessionBindingEvent *e*)
void valueUnbound(HhttpSessionBindingEvent *e*)

Aquí, *e* es el objeto que describe el vínculo.

La clase Cookie

La clase **Cookie** encapsula una cookie. Una *cookie* es almacenada en un cliente y contiene información del estado de la ejecución. Las cookies son valiosas para rastrear las actividades del usuario. Por ejemplo, consideremos que un usuario visita una tienda en línea.

Método	Descripción
Object getAttribute(String *attr*)	Devuelve el valor asociado con el nombre dado en *attr*. Devuelve **null** si attr no está presente en la sesión.
Enumeration getAttributeNames( )	Devuelve una enumeración con los nombres de los atributos asociados con la sesión.
long getCreationTime( )	Devuelve la hora (en milisegundos desde la media noche del 1 de enero de 1970 GMT) en que esta sesión fue creada.
String gemid( )	Devuelve el ID de la sesión.
long getLastAccessedTime( )	Devuelve la hora (en milisegundos desde la media noche del 1 de enero de 1970 GMT) cuando el cliente hizo por última vez una petición a la sesión.
void invalidate( )	Invalida la sesión y la elimina del contexto actual.
boolean isNew( )	Devuelve **true** si el servidor creó la sesión y ésta aún no ha sido accesada por el cliente.
void removeAttribute(String *attr*)	Elimina al atributo especificado por *attr* de la sesión.
void setAttribute(String *attr*, Object *val*)	Asocia el valor dado en *val* con el nombre de atributo pasado en *attr*.

TABLA 31-7 Métodos definidos por **HttpSession**

Una cookie puede almacenar el nombre del usuario, la dirección y más información. El usuario no necesita capturar esta información cada vez que visita la tienda.

Un servlet puede escribir una cookie en la máquina del cliente utilizando el método **addCookie( )** de la interfaz **HttpServletResponse**. Los datos de esta cookie se incluyen luego en el encabezado de la respuesta HTTP que se envía al navegador.

Los nombres y valores de una cookie se almacenan en la máquina del cliente. Parte de la información que es almacenada por cada cookie incluye lo siguiente:

- El nombre de la cookie
- El valor de la cookie
- La fecha de expiración de la cookie
- El dominio y ruta de la cookie

La fecha de expiración determina cuando la cookie será eliminada de la computadora del cliente. Si una fecha de expiración no se asigna explícitamente a la cookie, la cookie es eliminada cuando el navegador termine la sesión. De lo contrario, la cookie es almacenada en un archivo de la computadora del cliente.

El dominio y ruta de la cookie determinan cuándo la cookie es incluida en el encabezado de una petición HTTP. Si el usuario ingresa un URL cuyo dominio y ruta coinciden con los valores de dominio y ruta, la cookie será integrada en la petición.

Existe un constructor en la clase **Cookie**. El cual tiene la firma mostrada a continuación:

Cookie (String *nombre*, String *valor*)

Aquí, el nombre y valor de la cookie son proporcionados como argumentos al constructor. Los métodos de la clase **Cookie** se resumen en la Tabla 31-8.

Método	Descripción
Object clone( )	Devuelve una copia del objeto invocante.
String getComment( )	Devuelve un comentario.
String getDomain( )	Devuelve el dominio.
int getMaxAge( )	Devuelve la edad máxima (en segundos).
String getName( )	Devuelve el nombre.
String getPath( )	Devuelve la ruta.
String getSecure( )	Devuelve **true** si la cookie es segura. En caso contrario devuelve **false**.
String getValue( )	Devuelve el valor.
int getVersion( )	Devuelve la versión.
void setComment(String *c*)	Establece el valor del comentario con el valor *c*.
void setDomain(String *d*)	Establece el valor del dominio con el valor *d*.
void setMaxAge(int s)	Establece la edad máxima (en segundos) de la cookie al valor s. Este es el número de segundos después de los cuales la cookie es borrada.
void setPath(String *p*)	Establece el valor de la ruta con el valor *p*.
void setSecure(boolean s)	Establece la bandera de seguridad con el valor s.
void setValue(String *v*)	Establece el valor del campo value con el valor *v*.
void setVersion(int *v*)	Establece el valor de la versión con el valor *v*.

TABLA 31-8 Métodos definidos en **Cookie**

La clase HttpServlet

La Clase **HttpServlet** hereda de **GenericServlet**. Es utilizada comúnmente cuando se desarrollan servlets que reciben y procesan peticiones http. Los métodos de la clase **HttpServlet** se resumen en la Tabla 31-9.

La clase HttpSessionEvent

La clase **HttpSessionEvent** encapsula un evento de sesión. Esta clase hereda de **EventObject**, un objeto de esta clase se genera cuando un cambio ocurre en la sesión. El constructor de la clase tiene la siguiente firma:

```
HttpSessionEvent(HttpSession s)
```

Aquí, *s* es la fuente del evento.

HttpSessionEvent define un método, **getSession( )**, el cual se muestra a continuación:

```
HttpSession getSession( )
```

Devuelve la sesión en la cual ocurre el evento.

Método	Descripción
void doDelete(HttpServletRequest *req*, HttpServletResponse *res*) throws IOException, ServletException	Gestiona una petición DELETE vía HTTP.
void doGet(HttpServletRequest *req*, HttpServletResponse *res*) throws IOException, ServletException	Gestiona una petición GET vía HTTP.
void doHead(HttpServletRequest *req*, HttpServletResponse *res*) throws IOException, ServletException	Gestiona una petición HEAD vía HTTP.
void doOptions(HttpServletRequest *req*, HttpServletResponse *res*) throws IOException, ServletException	Gestiona una petición OPTIONS vía HTTP.
void doPost(HttpServletRequest *req*, HttpServletResponse *res*) throws IOException, ServletException	Gestiona una petición POST vía HTTP.
void doPut(HttpServletRequest *req*, HttpServletResponse *res*) throws IOException, ServletException	Gestiona una petición PUT vía HTTP.
void doTrace(HttpServletRequest *req*, HttpServletResponse *res*) throws IOException, ServletException	Gestiona una petición TRACE vía HTTP.
long getLastModified(HttpServletRequest *req*)	Devuelve la hora (en milisegundos desde la medianoche del 1 de enero de 1970 GMT) cuando el recurso solicitado fue modificado por última vez.
void service(HttpServletRequest *req*, HttpServletResponse *res*) throws IOException, ServletException	Es llamado por el servidor cuando se recibe una petición HTTP para el servlet. Los argumentos del método proporcionan acceso a la petición y la respuesta HTTP.

TABLA 31-9 Métodos definidos por **HttpServlet**

La clase HttpSessionBindingEvent

La clase **HttpSessionBindingEvent** hereda de **HttpSessionEvent**. Se genera un objeto de esta clase cuando un listener es enlazado o desenlazado a un valor en un objeto **HttpSession**. También se genera un objeto de esta clase cuando un atributo es enlazado o desenlazado. Los constructores de la clase son los siguientes:

HttpSessionBindingEvent(HttpSession *s*, String *nombre*)
HttpSessionBindingEvent(HttpSession *s*, String *nombre*, Object *valor*)

Aquí, *s* es la fuente del evento, y *nombre* es el nombre asociado con el objeto que está siendo enlazado o desenlazado. Si un atributo es enlazado o desenlazado, su valor es pasado en *valor*.

El método **getName( )** obtiene el nombre que está siendo enlazado o desenlazado. Este método se muestra a continuación:

String getName( )

El método **getSession( )**, mostrado enseguida, obtiene la sesión en la cual el listener está siendo enlazado o desenlazado:

HttpSession getSession( )

El método **getValue( )** obtiene el valor de un atributo que está siendo enlazado o desenlazado. El método tiene la siguiente firma:

Object getValue( )

Gestión de peticiones y respuestas de HTTP

La clase **HttpServlet** proporciona métodos especializados que gestionan los diversos tipos de peticiones HTTP. El programador comúnmente sobrescribe uno de esos métodos. Esos métodos son **doDelete( )**, **doGet( )**, **doHead( )**, **doOptions( )**, **doPost( )**, **doPut( )** y **doTrace( )**. Una descripción completa de los diferentes tipos de peticiones http está más haya del alcance de este libro. Sin embargo, las peticiones GET y POST son utilizadas comúnmente para trabajar con el elemento FORM de HTML. Por ello, esta sección presenta ejemplos de estas peticiones.

Gestión de peticiones tipo GET

Desarrollaremos ahora un servlet que gestiona peticiones GET. El servlet es invocado cuando una forma en una página Web es enviada al servlet. El ejemplo contiene dos archivos. Una página Web almacenada en el archivo **ColorGet.htm**, y un servlet almacenado en el archivo **ColorGetServlet.java**. El código HTML del archivo **ColorGet.htm** se muestra a continuación. En el código HTML se define una forma que contiene un elemento SELECT y un botón de envío. Note que el parámetro *action* de la forma especifica el URL del servlet que procesará la forma.

```
<html>
<body>
<center>
<form name="Form1"
  action = "http://localhost:8080/servlets-examples/servlet/ColorGetServlet">
<B>Color:</B>
<select name="color" size="1">
<option value="Rojo">Rojo</option>
<option value="Verde">Verde</option>
<option value="Azul">Azul</option>
</select>
<br><br>
<input type=submit value="Enviar">
</form>
</body>
</html>
```

El código fuente del servlet **ColorGetServlet.java** se muestra a continuación. El método **doGet( )** se sobrescribe para procesar cualquier petición HTTP GET que sea enviada al servlet. Se utiliza el método **getParameter( )** de **HttpServletRequest** para obtener la opción seleccionada por el usuario. Con base a dicho parámetro se genera una respuesta.

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

```
import java.servlet.*;
import javax.servlet.http.*;

public class ColorGetServlet extends HttpServlet {

  public void doGet(ServletRequest request, ServletResponse response)
throws ServletException, IOException {

    String color = request.getParameter("color");
    response.setContentType("text/html");
    PrintWriter pw = response.getWriter();
    pw.println( "<B>El color seleccionado fue: ");
    pw.println(color);
    pw.close();
  }
}
```

Compilemos el servlet y copiemos el archivo class al directorio apropiado y actualicemos el archivo **web.xml** como se describió antes. Luego, ejecutemos los siguientes pasos para probar el ejemplo:

1. Iniciamos Tomcat (si no está corriendo ya).
2. Desplegamos la página Web en el navegador.
3. Seleccionamos un color.
4. Finalmente se presiona el botón para enviar la página Web.

Después de seguir los pasos anteriores, el navegador desplegará una respuesta generada dinámicamente por el servlet.

Un comentario más: Los parámetros de una petición HTTP GET se incluyen como parte del URL que se envía al servidor. Por ejemplo si el usuario selecciona el color rojo y luego presiona el botón de envío, se genera la siguiente URL

```
http://localhost:8080/servlets-examples/servlet/ColorGetServlet?color=Rojo
```

Los caracteres a la derecha del signo de interrogación se conocen en conjunto como el *query-string*.

Gestión de peticiones tipo POST

Desarrollaremos ahora un servlet que gestiona peticiones POST. El servlet es invocado cuando una forma en una página Web es enviada al servlet. El ejemplo contiene dos archivos. Una página Web almacenada en el archivo **ColorPost.htm**, y un servlet almacenado en el archivo **ColorPostServlet.java.** El código HTML del archivo **ColorPost.htm** se muestra a continuación. El código es idéntico a **ColorGet.htm** excepto el parámetro method de la forma que explícitamente especifica el uso del método POST y el parámetro action ahora especifica el URL de un servlet diferente.

```
<html>
<body>
<center>
<form name = "Form1"
  method = "post"
  action = "http://localhost:8080/servlets-examples/servlet/ColorPostServlet">
<B>Color:</B>
```

```
<select name="color" size="1">
<option value="Rojo">Rojo</option>
<option value="Verde">Verde</option>
<option value="Azul">Azul</option>
</select>
<br><br>
<input type=submit value="Enviar">
</form>
</body>
</html>
```

El código fuente del servlet **ColorPostServlet.java** se muestra a continuación. El método **doPost( )** se sobrescribe para procesar cualquier petición HTTP POST que sea enviada al servlet. Se utiliza el método **getParameter( )** de **HttpServletRequest** para obtener la opción seleccionada por el usuario. Con base a dicho parámetro se genera una respuesta.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ColorPostServlet extends HttpServlet {

  public void doPost(ServletRequest request, ServletResponse response)
  throws ServletException, IOException {

    String color = request.getParameter("color");
    response.setContentType("text/html");
    PrintWriter pw = response.getWriter();
    pw.println( "<B>El color seleccionado fue: ");
    pw.println(color);
    pw.close();
  }
}
```

Compilemos el servlet y ejecutemos los mismos pasos descritos en la sección previa para probarlo.

NOTA Los parámetros para una petición HTTP POST no se incluyen como parte del URL que se envía al servidor Web. En este ejemplo, el URL enviado del navegador al servidor es `http://localhost:8080/servlets-examples/servlet/ColorPostServlet`
Los nombres de los parámetros y sus valores se envían en el cuerpo de la petición HTTP.

Uso de cookies

Ahora desarrollemos un servlet que ilustre como utilizar cookies. El servlet es invocado cuando el botón enviar de una forma en una página Web es presionado. El ejemplo contiene tres archivos que se explican a continuación:

Archivo	Descripción
AddCookie.htm	Permite al usuario especificar un valor para la cookie llamada **MiCookie**.
AddCookieServlet.java	Procesa el envío de **AddCookie.htm**.
GetCookiesServlet.java	Muestra los valores de las cookies.

El código fuente HTML para **AddCookie.htm** se muestra a continuación. Esta página contiene un campo de texto en el cual se escribe el valor a utilizar para la cookie. También contiene un botón de envío. Cuando el botón es presionado, el valor en el campo de texto es enviado a **AddCookieServlet** en la petición HTTP POST.

```
<html>
<body>
<center>
<form name="Form1"
  method="post"
  action = "http://localhost:8080/servlets-examples/servlet/AddCookieServlet">
<B>Escriba el valor a almacenar en MiCookie:</B>
<input type=textbox name="data" size=25 value="">
<input type=submit value="Enviar">
</form>
</body>
</html>
```

El código fuente de **AddCookieServlet.java** se muestra a continuación. Este servlet obtiene el valor del parámetro llamado"data", luego crea un objeto **Cookie** que tiene el nombre"MiCookie"y contiene el valor del parámetro"data". La cookie es añadida después al encabezado de la respuesta HTTP por una llamada al método **addCookie( )**. Finalmente aparece un mensaje en el navegador.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class AddCookieServlet extends HttpServlet {

  public void doPost(ServletRequest request,
  ServletResponse response)
  throws ServletException, IOException {

    // obtiene el parámetro de la petición HTTP
    String data = request.getParameter("data");

    // crea la cookie
    Cookie cookie = new Cookie("MiCookie", data);

    // añade la cookie al encabezado de la respuesta HTTP
    response.addCookie(cookie);

    // escribe la salida al navegador
    response.setContentType("text/html");
    PrintWriter pw = response.getWriter();
    pw.println( "<B>El valor: " + data + "ha sido asignado a MiCookie");
    pw.close();
  }
}
```

El código fuente de **GetCookiesServlet.java** se muestra a continuación. Este servlet invoca al método **getCookies( )** para leer cualquier cookie que esté incluida en la petición HTTP GET. Los nombres y valores de esas cookies se escriben en la respuesta HTTP. Observe que los métodos **getName( )** y **getValue( )** son invocados para obtener la información correspondiente.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class GetCookiesServlet extends HttpServlet {

  public void doGet(ServletRequest request, ServletResponse response)
  throws ServletException, IOException {

    // obtiene las cookies presentes en el encabezado de la petición http
    Cookie[] cookies = request.getCookies();

    // muestra las cookies
    response.setContentType("text/html");
    PrintWriter pw = response.getWriter();
    pw.println("<B>");
    for (int i = 0; i < cookies.length; i++) {
      String name = cookies[i].getName();
      String value = cookies[i].getValue();
      pw.println( "nombre: " + name + "; valor = " + value);
    }
    pw.close();
  }
}
```

Compilemos el servlet y copiemos el archivo class al directorio apropiado y actualicemos el archivo **web.xml** como se describió antes. Luego, ejecutemos los siguientes pasos para probar el ejemplo:

1. Iniciamos Tomcat (si no está corriendo ya).
2. Desplegamos **AddCookie.htm** en el navegador.
3. Escribimos un valor para **MiCookie**
4. Finalmente se presiona el botón para enviar la página Web.

Después de seguir los pasos anteriores, el navegador desplegará un mensaje dinámicamente por el servlet.

Después de lo anterior, utilizamos el navegador para acceder a esta dirección URL

```
http://localhost:8080/servlets-examples/servlet/GetCookiesServlet
```

Observe que el nombre y el valor de la cookie se muestran en el navegador.

En este ejemplo, no se asigna una fecha de expiración explícitamente a la cookie vía el método **setMaxAge( )** de la clase **Cookie**. Por consiguiente, la cookie expira cuando la sesión del navegador termina. Se puede experimentar utilizando **setMaxAge( )** y observaríamos que la cookie se almacena en el disco de la máquina del cliente.

Sesiones

HTTP es un protocolo que no gestiona información de su estado actual. Cada solicitud es independiente de la anterior. Sin embargo, en algunas aplicaciones, es necesario almacenar información del estado de la transferencia de manera que la información pueda ser recolectada a partir de varias interacciones entre el navegador y el servidor. Las sesiones proporcionan este mecanismo.

Una sesión puede ser creada utilizando el método **getSession( )** de **HttpServeletRequest**. El método regresa un objeto de tipo **HttpSession**. Este objeto puede almacenar un conjunto de vínculos que asocian nombres con objetos. Los métodos **setAttribute( )**, **getAttribute( )**, **getAttributeNames( )** y **removeAttribute( )** de la clase **HttpSession** administran esos vínculos. Es importante observar que el estado de una sesión es compartido por todos los servlets que están asociados con un cliente en particular.

El siguiente servlet ilustra cómo utilizar una sesión. El método **getSession( )** obtiene la sesión actual. Una nueva sesión se crea si actualmente no existe una ya creada. El método **getAttribute( )** se llama para obtener al objeto vinculado con el nombre"date". Este objeto es de tipo Date y encapsula la fecha y hora del último acceso a esta página (por supuesto no existe éste valor la primera vez que la página es ingresada). Luego se crea un objeto de tipo **Date** que encapsula la fecha y hora. El método **setAttribute( )** es invocado para vincular al nombre"date" con el objeto de tipo Date creado.

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class DateServlet extends HttpServlet {

  public void doGet(HttpServletRequest request, HttpServletResponse response)
  throws ServletException, IOException {

    // Obtiene la sesión
    HttpSession hs = request.getSession(true);

    // Obtiene al flujo de salida
    response.setContentType("text/html");
    PrintWriter pw = response.getWriter();
    pw.print("<B>");

    // Muestra la fecha y hora del último acceso
    Date date = (Date)hs.getAttribute("date");
    if (date != null) {
      pw.print("Último acceso: " + date + "<BR>");
    }

    // Muestra la hora y fecha actual
    date = new Date();
    hs.setAttribute("date", date);
    pw.println("Fecha actual:" + date);
  }
}
```

Cuando se ingresa por primera vez al servlet, el navegador despliega una línea con la fecha y hora actual. En invocaciones subsecuentes, se despliegan dos líneas. La primera línea muestra la fecha y hora del último acceso. La segunda línea muestra la fecha y hora actuales.

IV PARTE

Aplicaciones en Java

CAPÍTULO 32

Applets y servlets aplicados en la solución de problemas

Pese a todas las grandes y sofisticadas aplicaciones, tales como procesadores de palabras, bases de datos y paquetes contables que dominan gran parte del panorama computacional, se ha mantenido una clase de programas que son tanto populares como pequeños. Dichos programas ejecutan cálculos financieros, tal como el pago regular de préstamos, el valor futuro de una inversión, o el saldo de un préstamo. Ninguno de estos cálculos es complicado o requieren demasiado código, pero proporcionan información que es muy útil.

Como sabemos, Java fue diseñado inicialmente para soportar la creación de pequeños programas portables. Originalmente, estos programas tomaron la forma de applets, pero pocos años más tarde se agregaron los servlets. Recuerde que los *applets* corren en la máquina local, dentro de un navegador y los *servlets* se ejecutan en el servidor. Dado su pequeño tamaño, muchos de los cálculos financieros comunes son realizados por applets y servlets. Además, incluir un applet/servlet financiero en una página Web es una amabilidad que muchos usuarios apreciarán. Un usuario regresará una y otra vez a la página que ofrece el cálculo que él desea.

Este capítulo desarrolla varios de applets que ejecutan los siguientes cálculos financieros:

- Pagos regulares sobre un préstamo
- Saldo promedio de un préstamo
- Futuro valor de una inversión
- Inversión inicial requerida para atender un valor futuro deseado
- Anualidad de una inversión
- Inversión necesaria para una anualidad deseada

El capítulo finaliza mostrando como convertir los applets financieros en servlets.

Calcular los pagos de un préstamo

Posiblemente el cálculo financiero más popular es el que calcula los pagos regulares de un préstamo, tal como el préstamo para un auto o para una casa. Los pagos sobre un préstamo se calculan usando la siguiente formula:

$$\text{Payment} = (intRate * (principal / payPerYear)) / (1 - ((intRate / payPerYear) + 1)^{-payPerYear * numYenrs})$$

Donde *initRate* especifica la tasa de interés, *principal* contiene el balance inicial, *payPerYear* especifica el número de pagos por año, y *numYears* especifica la duración del préstamo en años.

El siguiente applet llamado **RegPay** utiliza la formula anterior para calcular los pagos de un préstamo con información ingresada por el usuario. Como todos los applets en este capítulo, **RegPay** es un applet basado en Swing. Esto significa que extiende de la clase **JApplet** y utiliza las clases de Swing para construir la interfaz gráfica de usuario. Note que también implementa la interfaz **ActionListener**.

```
// Un ejemplo de applet que calcula los pagos de un préstamo.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.text.*;
/*
  <applet code="RegPay" width=320 height=200>
  </applet>
*/

public class RegPay extends JApplet
  implements ActionListener {

  JTextField amountText, paymentText, periodText, rateText;
  JButton doIt;

  double principal; // monto original
  double intRate; // tasa de interés
  double numYears; // duración del préstamo

  /* Número de pagos por año. Se podría permitir
  que este valor sea definido por el usuario. */
  final int payPerYear = 12;

  NumberFormat nf;

  public void init() {
    try {
      SwingUtilities.invokeAndWait(new Runnable() {
        public void run() {
          makeGUI(); // inicializa la GUI
        }
      });
    } catch(Exception exc) {
      System.out.println("No se puede crear debido a "+ exc);
  }
}
```

```
// Crea e inicializa el GUI.
private void makeGUI() {

  // Uso de un GridBagLayout.
  GridBagLayout gbag = new GridBagLayout();
  GridBagConstraints gbc = new GridBagConstraints();
  setLayout(gbag);

  JLabel heading = new
         JLabel("Calcula los pagos mensuales de un préstamo");

  JLabel amountLab = new JLabel("Capital ");
  JLabel periodLab = new JLabel("Años ");
  JLabel rateLab = new JLabel("Tasa de Interés ");
  JLabel paymentLab = new JLabel("Pagos mensuales ");

  amountText = new JTextField(10);
  periodText = new JTextField(10);
  paymentText = new JTextField(10);
  rateText = new JTextField(10) ;

  // Campo de pagos solo para desplegar información.
  paymentText.setEditable(false) ;

  doIt = new JButton("Calcular");

  // Define el acomodo de los componentes
  gbc.weighty = 1.0; // usa un peso de fila igual a 1
  gbc.gridwidth = GridBagConstraints.REMAINDER;
  gbc.anchor = GridBagConstraints.NORTH;
  gbag.setConstraints(heading, gbc);

  // Alinea la mayoría de los componentes a la derecha
  gbc.anchor = GridBagConstraints.EAST;

  gbc.gridwidth = GridBagConstraints.RELATIVE;
  gbag.setConstraints(amountLab, gbc);
  gbc.gridwidth = GridBagConstraints.REMAINDER;
  gbag.setConstraints(amountText, gbc);

  gbc.gridwidth = GridBagConstraints.RELATIVE;
  gbag.setConstraints(periodLab, gbc);
  gbc.gridwidth = GridBagConstraints.REMAINDER;
  gbag.setConstraints(periodText, gbc);

  gbc.gridwidth = GridBagConstraints.RELATIVE;
  gbag.setConstraints(rateLab, gbc);
  gbc.gridwidth = GridBagConstraints.REMAINDER;
  gbag.setConstraints(rateText, gbc);

  gbc.gridwidth = GridBagConstraints.RELATIVE;
  gbag.setConstraints(paymentLab, gbc);
  gbc.gridwidth = GridBagConstraints.REMAINDER;
  gbag.setConstraints(paymentText, gbc);

  gbc.anchor = GridBagConstraints.CENTER;
  gbag.setConstraints(doIt, gbc);
```

```
    // Agrega todos los componentes.
    add(heading);
    add(amountLab);
    add(amountText);
    add(periodLab);
    add(periodText);
    add(rateLab);
    add(rateText);
    add(paymentLab);
    add(paymentText);
    add(doIt);

    // Registra los listeners para recibir eventos de acción.
    amountText.addActionListener(this);
    periodText.addActionListener(this);
    rateText.addActionListener(this);
    doIt.addActionListener(this);

    // formato numérico
    nf = NumberFormat.getInstance();
    nf.setMinimumFractionDigits(2);
    nf.setMaximumFractionDigits(2);
  }

  /* El usuario presionó Enter sobre un campo de texto o
    presionó Calcular. Muestra el resultado si todos los
    campos están completos. */
  public void actionPerformed(ActionEvent ae) {
    double result = 0.0;

    String amountStr = amountText.getText();
    String periodStr = periodText.getText();
    String rateStr = rateText.getText();

    try {
      if(amountStr.length() != 0 &&
      periodStr.length() != 0 &&
      rateStr.length() != 0) {

      principal = Double.parseDouble(amountStr);
      numYears = Double.parseDouble(periodStr);
      intRate = Double.parseDouble(rateStr) / 100;

      result = compute();

      paymentText.setText(nf.format(result)) ;
    }

    showStatus(" "); // borra cualquier mensaje de error previo
      } catch (NumberFormatException exc) {
        showStatus("Datos Inválidos");
        paymentText.setText("");
      }
    }
```

```
  //Calcula el pago del préstamo
  double compute() {
    double numer;
    double denom;
    double b,e;

    numer = intRate * principal / payPerYear;

    e = -(payPerYear * numYears);
    b = (intRate / payPerYear) + 1.0;

    denom = 1.0 - Math.pow(b, e);

    return numer / denom;
  }
}
```

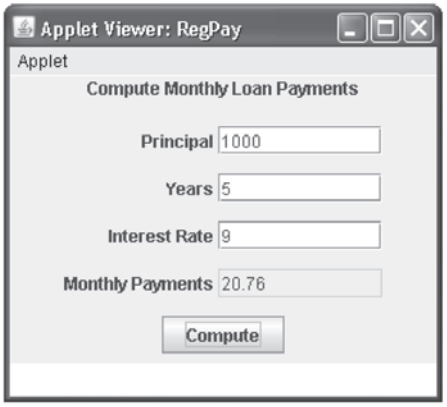
El applet producido por este programa se muestra en la figura 32-1. Para utilizar el applet, simplemente introduzca el monto del préstamo, la longitud del préstamo en años, y la tasa de interés. Se asume que los pagos serán mensuales. Una vez que la información se ingresa, presionamos el botón Calcular para calcular el monto de los pagos mensuales.

Las siguientes secciones examinan el código de **RegPay** a detalle. Debido a que todos los applets en este capítulo utilizan la misma estructura básica, muchas de las explicaciones dadas aquí también aplican para los otros applets.

Las variables de la clase

La clase **RegPay** comienza con la declaración de diversas variables de instancia que mantienen las referencias a los campos de texto dentro de los cuales el usuario ingresará la información del préstamo. A continuación, se declara la variable **doIt** la cual mantiene la referencia al botón Calcular.

FIGURA 32-1
El applet **RegPay**



Después, la clase **RegPay** declara tres variables **double** que almacenan los valores del préstamo. El valor del campo capital se almacena en **principal**, la tasa de interés se almacena en **intRate**, y la longitud del préstamo en años se almacena en **numYears**. Estos valores son ingresados por el usuario a través de los campos de texto. A continuación, la constante entera **payPerYear** es declarada e inicializada a 12. De esta forma, el número de pagos por año está codificado para ser mensual, debido a que ésta es la forma en que la mayoría de los préstamos se pagan. Como los comentarios sugieren, se podría permitir al usuario ingresar este valor, pero hacerlo haría necesario la adición de otro campo de texto.

La última variable de instancia declarada por **RegPay** es **nf**, una referencia a un objeto de tipo **NumberFormat**, el cual describe el formato numérico utilizado en las salidas. La clase **NumberFormat** está contenida en el paquete **java.text**. Aunque existen otras formas de dar formato a la salida, como utilizar la clase **Formatter**, **NumberFormat** es una buena elección en este caso, debido a que el mismo formato es utilizada con regularidad, y este formato puede ser definido una vez al inicio del programa. Los applets financieros también ofrecen una buena oportunidad para demostrar su uso.

El método init( )

Como en todos los applets, el método **init( )** es llamado cuando el applet comienza su ejecución. Este método simplemente invoca al método **makeGUI( )** en el hilo gestor de eventos. Como se explicó en el Capítulo 29, los applets basados en Swing deben construir e interactuar con los componentes gráficos sólo a través del hilo gestor de eventos.

El método makeGUI( )

El método **makeGUI( )** crea la interfaz de usuario para el applet. Este método ejecuta las siguientes labores:

1. Cambia el gestor de organización por un **GridBagLayout**.
2. Instancia a los diversos componentes gráficos.
3. Añade los componentes a la cuadrícula.
4. Añade listeners de acción a los componentes.

Veamos al método **makeGUI( )** línea por línea. El método comienza con estas líneas de código:

```
// Uso de un GridBagLayout.
GridBagLayout gbag = new GridBagLayout();
GridBagConstraints gbc = new GridBagConstraints();
setLayout(gbag);
```

Estas líneas crean un organizador de tipo **GridBagLayout** que será utilizado por el applet (para más detalles sobre el uso del **GridBagLayout**, vea el Capítulo 24). Utilizaremos **Grid BagLayout** porque permite tener un mayor control sobre la posición de controles en el applet.

A continuación, **makeGUI( )** crea algunas etiquetas, campos de texto y el botón Calcular, como se muestra a continuación:

```
JLabel heading = new JLabel("Calcula los pagos mensuales de un préstamo");
JLabel amountLab = new JLabel("Capital ");
JLabel periodLab = new JLabel("Años ");
JLabel rateLab = new JLabel("Tasa de Interés ");
JLabel paymentLab = new JLabel("Pagos mensuales ");
```

```
amountText = new JTextField(10);
periodText = new JTextField(10);
paymentText = new JTextField(10);
rateText = new JTextField(10) ;

// Campo de pagos sólo para desplegar información.
paymentText.setEditable(false) ;

doIt = new JButton("Calcular");
```

Note que el campo de texto que muestra los pagos mensuales está definido como sólo lectura con la llamada al método **setEditable(false)**. Esto causa que el campo se vuelva gris y no se pueda ingresar ningún texto en ese campo. Sin embargo, el contenido del campo puede ser modificado llamando al método **setText( )**. De esta forma, cuando la edición se deshabilita en un **JTextField**, el campo puede ser usado para mostrar el texto, aunque el texto no pueda ser cambiado por el usuario.

Luego, se definen las restricciones de la cuadrícula para cada componente con la siguiente secuencia de código:

```
// Define el acomodo de los componentes
gbc.weighty = 1.0; // usa un peso de fila igual a 1
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbc.anchor = GridBagConstraints.NORTH;
gbag.setConstraints(heading, gbc);

// Alinea la mayoría de los componentes a la derecha
gbc.anchor = GridBagConstraints.EAST;

gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(amountLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(amountText, gbc);

gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(periodLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(periodText, gbc);

gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(rateLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(rateText, gbc);

gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(paymentLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(paymentText, gbc);

gbc.anchor = GridBagConstraints.CENTER;
gbag.setConstraints(doIt, gbc);
```

Aunque esto parece un poco complicado a primera vista, no lo es realmente. Sólo recuerde que cada fila se especifica por separado. La secuencia funciona de la siguiente forma. Primero, el peso de cada fila, especificada en **gbc.weighty**, se fija en 1. Esto le dice al organizador que distribuya el espacio extra equitativamente cuando haya más espacio vertical que el que se requiere para mantener los componentes. A continuación, **gbc.gridwidth** se define como **REMAINDER**, y **gbc.anchor** se define como **NORTH**. La etiqueta referenciada por la variable

heading se agrega llamando al método **setConstraints( )** sobre **gbag**. Esta secuencia define la ubicación del **encabezado** en la parte superior de la cuadrícula y concede el resto a la fila. Por lo tanto, después de que esta secuencia se ejecuta, el encabezado estará en la parte superior de la ventana y sobre una misma fila.

A continuación, se agregan los cuatro campos de texto y sus etiquetas. Primero, **gbc.anchor** se define como **EAST**. Esto causa que cada componente sea alineado a la derecha. Luego, **gbc.gridWidth** se define como **RELATIVE** y se añade la etiqueta. Luego, **gbc.gridWidth** se define como **REMAINDER** y se añade el campo de texto. De esta forma, cada par formado por un campo de texto y una etiqueta ocupa una fila. Este proceso se repite hasta que los cuatro campos de texto y sus etiquetas han sido agregados. Finalmente, el botón Calcular se agrega en el centro.

Después de que la cuadrícula ha sido definida, los componentes son agregados a la ventana con el código siguiente:

```
// Agrega todos los componentes.
add(heading);
add(amountLab);
add(amountText);
add(periodLab);
add(periodText);
add(rateLab);
add(rateText);
add(paymentLab);
add(paymentText);
add(doIt);
```

A continuación, se registran los listeners de acción para los tres campos de texto y el botón Calcular, como se muestra a continuación:

```
// Registra los listeners para recibir eventos de acción.
amountText.addActionListener(this);
periodText.addActionListener(this);
rateText.addActionListener(this);
doIt.addActionListener(this);
```

Finalmente, se obtiene un objeto **NumberFormat** y se define el formato con dos dígitos decimales:

```
// formato numérico
nf = NumberFormat.getInstance();
nf.setMinimumFractionDigits(2);
nf.setMaximumFractionDigits(2);
```

La llamada al método de fábrica **getInstance( )** obtiene un objeto **NumberFormat** adecuado para la localidad por omisión. Las llamadas a los métodos **setMinimumFractionDigits( )** y **setMaximumFractionDigits( )** definen el mínimo y máximo número de dígitos que serán mostrados. Debido a que ambos son definidos a dos, esto asegura que dos decimales serán siempre visibles.

El método actionPerformed( )

El método **actionPerformed( )** es llamado en cualquier momento que el usuario presiona ENTER, cuando está en un campo de texto o da clic al botón Calcular. Este método ejecuta tres funciones principales: obtiene la información del préstamo escrita por el usuario, llama al método **compute( )** para calcular los pagos del préstamo y muestra el resultado. Examinemos ahora al método **actionPerformed( )** línea por línea.

Después de declarar la variable **result**, el método **actionPerformed( )** comienza por obtener las cadenas de las tres entradas del usuario utilizando la siguiente secuencia:

```
String amountStr = amountText.getText();
String periodStr = periodText.getText();
String rateStr = rateText.getText();
```

Después, comienza un bloque **try** que verifica que los tres campos contengan información como se muestra aquí:

```
try {
  if(amountStr.length() != 0 &&
    periodStr.length() != 0 &&
    rateStr.length() != 0) {
```

Recuerde que el usuario debe ingresar la cantidad de préstamo original, la duración del préstamo en años, y la tasa de interés. Si los tres campos de texto contienen información, entonces la longitud de cada cadena será mayor a cero.

Si el usuario ha ingresado todo los datos del préstamo, entonces lo valores numéricos correspondientes a esas cadenas se almacenan en la variable de instancia apropiada. Después, se llama al método **compute( )** para calcular los pagos del préstamo, y el resultado se muestra en el campo de texto de sólo lectura declarado como **paymentText**, como se muestra a continuación:

```
principal = Double.parseDouble(amountStr);
numYears = Double.parseDouble(periodStr);
intRate = Double.parseDouble(rateStr) / 100;

result = compute();

paymentText.setText(nf.format(result)) ;
```

Observe la llamada a **nf.format(result)**. La cual provoca que el valor en **result**, sea formateado como se especificó previamente (con dos dígitos decimales) y la cadena resultante sea devuelta. Esta cadena es utilizada como el texto que se coloca en el **JTextField** especificado por **paymentText**.

Si el usuario ha ingresado un valor no numérico dentro de un campo de texto, entonces **Double.parseDouble( )** lanzará una excepción de tipo **NumberFormatException**. Si esto ocurre, un mensaje de error será mostrado sobre la línea de estado y el campo de texto Payment será limpiado, como se muestra aquí:

```
  showStatus(" "); // borra cualquier mensaje de error previo
} catch (NumberFormatException exc) {
  showStatus("Datos Inválidos");
  paymentText.setText("");
}
```

De otra forma, cualquier error reportado previamente se borra.

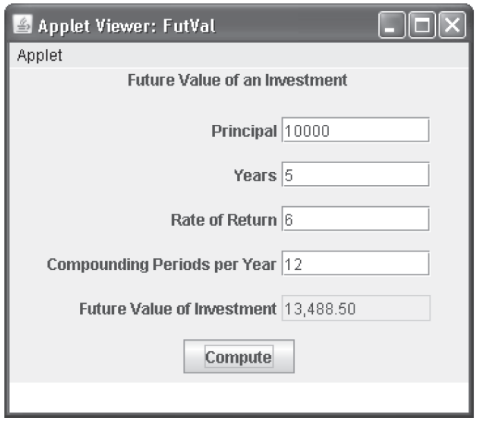
El método compute( )

El cálculo de los pagos del préstamo tienen lugar en el método **compute( )**. Este método implementa la fórmula mostrada anteriormente y opera sobre los valores **principal**, **intRate**, **numYears** y **payPerYear**. Finalmente devuelve el resultado.

---

*NOTA El esquema básico utilizado por **RegPay** es el mismo que utilizan todos los applets mostrados en este capítulo.*

FIGURA 32-2
El applet **FutVal**



Calcular el valor futuro de una inversión

Otro cálculo financiero popular es el que encuentra el valor futuro de una inversión dada la inversión inicial, la tasa de retorno, el número de periodos calculados por año, y el número de años en que la inversión se mantiene. Por ejemplo, podría querer saber cuánto se ganaría en 12 años si actualmente la cuenta tiene $98,000 y tiene un promedio anual de retorno del 6 por ciento. El applet **FutVal** desarrollado aquí proveerá la respuesta.

Para calcular el valor futuro, se utiliza la siguiente fórmula:

Valor Futuro = principal * ((rateOfRet / compPerYear) + 1) $^{\text{compPerYear * numYears}}$

Donde *rateOfRet* especifica la tasa de retorno, principal contiene el valor inicial de la inversión, *compPerYear* especifica el número de periodos compuestos por año, y *numYears* especifica la longitud de la inversión en años. Si se utiliza una tasa anualizada de retorno para *rateOfRet*, entonces el número de periodos comprendidos es 1.

El siguiente applet llamado **FutVal** utiliza la fórmula precedente para calcular el valor futuro de una inversión. El applet producido por este programa se muestra en la Figura 32-2. Aparte de las diferencias computacionales dentro del método **compute( )**, el applet es muy similar en operación al applet **RegPay** descrito en la sección anterior.

```
// Un ejemplo de applet que calcula el valor futuro de una inversión.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.text.*;
/*
  <applet code="FutVal" width=380 height=240>
  </applet>
*/
```

```
public class FutVal extends JApplet
  implements ActionListener {

  JTextField amountText, futvalText, periodText, rateText, compText;

  JButton doIt;
  double principal; // inversión inicial
  double rateOfRet; // tasa de retorno
  double numYears; // duración de la inversión en años
  int compPerYear; // número de periodos por año

  NumberFormat nf;

  public void init() {
    try {
      SwingUtilities.invokeAndWait(new Runnable () {
        public void run() {
          makeGUI(); // inicializa el GUI
        }
      });
    } catch (Exception exc) {
      System.out.println("No se puede crear debido a: "+ exc);
    }
  }

  // Crea e inicializa el GUI.
  private void makeGUI() {

    // Utiliza un GridBagLayout.
    GridBagLayout gbag = new GridBagLayout();
    GridBagConstraints gbc = new GridBagConstraints();
    setLayout(gbag);

    JLabel heading = new JLabel("Valor futuro de una Inversión");

    JLabel amountLab = new JLabel("Capital ");
    JLabel periodLab = new JLabel("Años ");
    JLabel rateLab = new JLabel("Tasa de Retorno ");
    JLabel futvalLab = new JLabel("Valor futuro de inversión ");
    JLabel compLab = new JLabel("Periodos por Año ");

    amountText = new JTextField(10);
    periodText = new JTextField(10);
    futvalText = new JTextField(10);
    rateText = new JTextField(10);
    compText = new JTextField(10);

    // Campo para mostrar el valor futuro
    futvalText.setEditable(false);

    doIt = new JButton("Calcular");

    // Acomodo del GridBagLayout
    gbc.weighty = 1.0; // utiliza un peso para el renglón igual a 1
    gbc.gridwidth = GridBagConstraints.REMAINDER;
    gbc.anchor = GridBagConstraints.NORTH;
    gbag.setConstraints(heading, gbc);
```

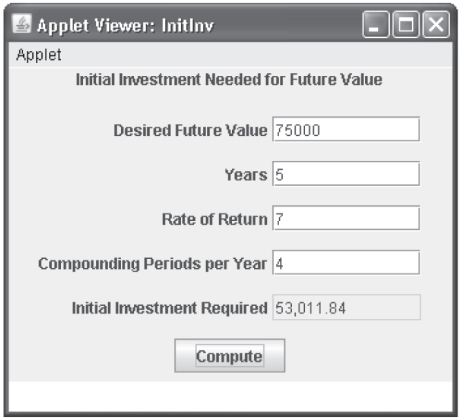
```
// Alinea la mayoría de los componentes a la derecha
gbc.anchor = GridBagConstraints.EAST;

gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(amountLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(amountText, gbc);

gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(periodLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(periodText, gbc);

gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(rateLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(rateText, gbc);

gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(compLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(compText, gbc);

gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(futvalLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(futvalText, gbc);

gbc.anchor = GridBagConstraints.CENTER;
gbag.setConstraints(doIt, gbc);

add(heading);
add(amountLab);
add(amountText);
add(periodLab);
add(periodText);
add(rateLab);
add(rateText) ;
add(compLab) ;
add(compText) ;
add(futvalLab);
add(futvalText);
add(doIt) ;

// registra los listeners para recibir eventos de acción
amountText.addActionListener(this);
periodText.addActionListener(this);
rateText.addActionListener(this);
compText.addActionListener(this);
doIt.addActionListener(this);

// Crea un formato numérico
nf = NumberFormat.getInstance();
nf.setMinimumFractionDigits(2);
```

```
    nf.setMaximumFractionDigits(2);
  }

  /* El usuario presiona Enter en un campo de texto o
    presiona el botón "Calcular". Muestra el resultado si todos los
    campos están completos */
  public void actionPerformed(ActionEvent ae) {
    double result = 0.0;

    String amountStr = amountText.getText();
    String periodStr = periodText.getText();
    String rateStr = rateText.getText();
    String compStr = compText.getText();

    try {
      if(amountStr.length() != 0 &&
        periodStr.length() != 0 &&
        rateStr.length() != 0 &&
        compStr.length() != 0) {

        principal = Double.parseDouble(amountStr);
        numYears = Double.parseDouble(periodStr);
        rateOfRet = Double.parseDouble(rateStr) / 100;
        compPerYear = Integer.parseInt(compStr);

        result = compute();

        futvalText.setText(nf.format(result));
      }

      showStatus(""); // borra cualquier error anterior
    } catch (NumberFormatException exc) {
      showStatus("Datos Inválidos");
      futvalText.setText("") ;
    }
  }

  // Calcula el valor futuro.
  double compute () {
    double b, e;

    b = (1 + rateOfRet/compPerYear);
    e = compPerYear * numYears;

    return principal * Math.pow(b, e);
  }
}
```

Calcular la inversión inicial requerida para alcanzar un valor futuro

Algunas veces se deseará saber que tan grande debe ser una inversión inicial para lograr algún valor futuro. Por ejemplo, si se está ahorrando para la educación universitaria de los hijos, sabrá que se requieren $75 000 en cinco años, ¿cuánto dinero se necesita invertir al 7 por ciento para alcanzar el objetivo? El applet **InitInv** desarrollado aquí puede responder esa pregunta.

FIGURA 32-3
El applet **InitInv**



La fórmula para calcular la inversión inicial se muestra a continuación:

Inversión Inicial = *targetValue* / (((*rateOfRet* / *compPerYear*) + 1) $^{compPerYear * numYears}$)

Donde *rateOfRet* especifica la tasa de retorno, *targetValue* contiene el balance inicial, *compPerYear* especifica el número de periodos por año, y *numYear* especifica la duración de la inversión en años. Si se utiliza una tasa de retorno anualizada para *rateOfRet*, entonces el número de periodos es 1.

El siguiente applet llamado **InitInv** utiliza la fórmula anterior para calcular la inversión inicial requerida para alcanzar el valor futuro deseado. El applet producido por este programa se muestra en la Figura 32-3.

```
/* Un ejemplo de applet que calcula el valor inicial de una inversión
   necesario para un valor futuro específico*/
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.text.*;
/*
  <applet code="InitInv" width=340 height=240>
  </applet>
*/

public class InitInvl extends JApplet
  implements ActionListener {

  JTextField targetText, initialText, periodText,
             rateText, compText;
  JButton doIt;

  double target; // inversión inicial
  double rateOfRet; // tasa de retorno
  double numYears; // duración de la inversión en años
  int compPerYear; // número de periodos por año

  NumberFormat nf;
```

```
  public void init() {
    try {
      SwingUtilities.invokeAndWait(new Runnable() {
        public void run() {
          makeGUI(); // inicializar la interfaz gráfica
        }
      });
    } catch(Exception exc) {
      System.out.println("No se puede crear la aplicación debido a "+ exc);
    }
  }

// Crea e inicializa la interfaz gráfica
private void makeGUI() {

    // Utiliza un GridBagLayout.
    GridBagLayout gbag = new GridBagLayout();
    GridBagConstraints gbc = new GridBagConstraints();
    setLayout(gbag);

    JLabel heading = new
           JLabel("Inversión inicial requerida para " +
                  "un valor futuro") ;

    JLabel targetLab = new JLabel("Valor futuro deseado ");
    JLabel periodLab = new JLabel("Años");
    JLabel rateLab = new JLabel ("Tasa de retorno ");
    JLabel compLab =
               new JLabel(" Periodos por año ");
    JLabel initialLab =
               new JLabel("Inversión inicial requerida ");

    targetText = new JTextField(10);
    periodText = new JTextField(10);
    initialText = new JTextField(10);
    rateText = new JTextField(10);
    compText = new JTextField(10);

    // El campo de valor inicial, se utiliza sólo
    para mostrar el resultado inicial
    initialText.setEditable(false);

    doIt = new JButton("Calcular");

    // Define el GridBagLayour.
    gbc.weighty = 1.0; // utiliza una fila con peso 1

    gbc.gridwidth = GridBagConstraints.REMAINDER;
    gbc.anchor = GridBagConstraints.NORTH;
    gbag.setConstraints(heading, gbc);

    // Alinea la mayoría de los componentes a la derecha
    gbc.anchor = GridBagConstraints.EAST;

    gbc.gridwidth = GridBagConstraints.RELATIVE;
    gbag.setConstraints(targetLab, gbc);
    gbc.gridwidth = GridBagConstraints.REMAINDER;
    gbag.setConstraints(targetText, gbc);
```

```
        gbc.gridwidth = GridBagConstraints.RELATIVE;
        gbag.setConstraints(periodLab, gbc);
        gbc.gridwidth = GridBagConstraints.REMAINDER;
        gbag.setConstraints(periodText, gbc);

        gbc.gridwidth = GridBagConstraints.RELATIVE;
        gbag.setConstraints(rateLab, gbc);
        gbc.gridwidth = GridBagConstraints.REMAINDER;
        gbag.setConstraints(rateText, gbc);

        gbc.gridwidth = GridBagCorrstraints.RELATIVE;
        gbag.setConstraints(compLab, gbc);
        gbc.gridwidth = GridBagConstraints.REMAINDER;
        gbag.setConstraints(compText, gbc);

        gbc.gridwidth = GridBagConstraints.RELATIVE;
        gbag.setConstraints(initialLab, gbc);
        gbc.gridwidth = GridBagConstraints.REMAINDER;
        gbag.setConstraints(initialText, gbc);

        gbc.anchor = GridBagConstraints.CENTER;
        gbag.setConstraints(doIt, gbc);

        // Agrega todos los componentes.
        add(heading);
        add(targetLab);
        add(targetText);
        add(periodLab);
        add(periodText);
        add(rateLab);
        add(rateText);
        add(compLab);
        add(compText);
        add(initialLab);
        add(initialText);
        add(doIt);

        // Registra los listener para recibir eventos de acción
        targetText.addActionListener(this);
        periodText.addActionListener(this);
        rateText.addActionListener(this);
        compText.addActionListener(this);
        doIt.addActionListener(this);

        // Crea un formato numérico.
        nf = NumberFormat.getInstance();
        nf.setMinimumFractionDigits(2);
        nf.setMaximumFractionDigits(2);
    }

    /* El usuario presionó Enter en un campo de texto
       o presionó el botón Calcular. Muestra el resultado si
       todos los campos están completos. */
```

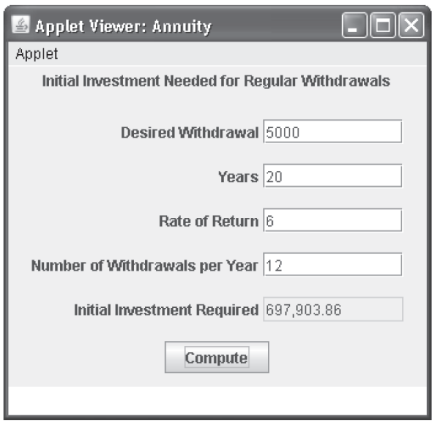
```
  public void actionPerformed(ActionEvent ae) {
    double result = 0.0;

    String targetStr = targetText.getText();
    String periodStr = periodText.getText();
    String rateStr = rateText.getText() ;
    String compStr = compText.getText();

    try {
      if(targetStr.length() != 0 &&
        periodStr.length() != 0 &&
        rateStr.length() != 0 &&
        compStr.length() != 0) {

        targetValue = Double.parseDouble(targetStr);
        numYears = Double.parseDouble(periodStr);
        rateOfRet = Double.parseDouble(rateStr) / 100;
        compPerYear = Integer.parseInt(compStr);

        result = compute();

        initialText.setText(nf.format(result));
      }

      showStatus(" "); // borra cualquier mensaje de error previo
    } catch (NumberFormatException exc) {
      showStatus("Datos Inválidos");
      initialText.setText("");
    }
  }

  // Calcula la inversión inicial.
  double compute() {
    double b, e;

    b = (1 + rateOfRet/compPerYear);
    e = compPerYear * numYears;

    return targetValue / Math.pow(b, e);
  }
}
```

Calcular la inversión inicial necesaria para una anualidad deseada

Otro cálculo financiero común es calcular el monto de dinero que se debe invertir para obtener una anualidad deseada, en retiros mensuales. Por ejemplo, podría pensar que se necesitará $50,000 por mes cuando se jubile y que será necesaria esa cantidad por 20 años. La pregunta es ¿cuánto se necesitará invertir para asegurar la anualidad? La respuesta puede ser encontrada utilizando la siguiente fórmula:

$$\text{Inversión Inicial} = ((regWD * wdPerYear) / rateOfRet) * (1 - (1 / (rateOfRet / wdPerYear + 1)^{wdPerYear * numYears}))$$

FIGURA 32-4
El applet **Annuity**



Donde *rateOfRet* especifica la tasa de retorno, *regWD* contiene el retiro regular deseado, *wdPerYear* especifica el número de retiros por año, y *numYears* especifica la duración de la anualidad en años.

El applet **Annuity** mostrado aquí calcula la inversión inicial requerida para producir la anualidad deseada. El applet producido por este programa se muestra en la Figura 32-4.

```
/*  Un ejemplo de applet que calcula la inversión inicial necesaria para
    una anualidad deseada. En otras palabras, encuentra el monto inicial
    necesario que permite retiros regulares de un monto deseado en un
    periodo de tiempo*/
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.text.*;
/*
  <applet code="Annuity" width=340 height=260>
  </applet>
*/

public class Annuity extends JApplet
  implements ActionListener {

  JTextField regWDText, initialText, periodText,
             rateText, numWDText;
  JButton doIt;

  double regWDAmount; // monto de cada retiro
  double rateOfRet; // tasa de retorno
  double numYears; // duración de la inversión en años
  int numPerYear; // número de retiros por año

  NumberFormat nf;
```

```
public void init() {
  try {
    SwingUtilities.invokeAndWait(new Runnable () {
      public void run() {
        makeGUI(); // inicializa el GUI
      }
    });
  } catch (Exception exc) {
    System.out.println("No se puede crear la aplicación debido a "+ exc);
  }
}

// Crea e inicializa la interfaz gráfica
private void makeGUI() {

  // Utiliza un GridBagLayout
  GridBagLayout gbag = new GridBagLayout();
  GridBagConstraints gbc = new GridBagConstraints();
  setLayout(gbag);

  JLabel heading = new
         JLabel("Inversión necesaria para" +
                "Retiros regulares");

  JLabel regWDLab = new JLabel("Retiros deseados");
  JLabel periodLab = new JLabel ("Años");
  JLabel rateLab = new JLabel("Tasa de Retorno");
  JLabel numWDLab =
           new JLabel ("Número de retiros por año");
  JLabel initialLab =
           new JLabel("Inversión inicial requerida");

  regWDText = new JTextField(10);
  periodText = new JTextField(10);
  initialText = new JTextField(10);
  rateText = new JTextField(10);
  numWDText = new JTextField(10);

  // campo para desplegar resultados
  initialText.setEditable(false);

  doIt = new JButton("Calcular");

  // Define el GridBagLayout

  gbc.weighty = 1.0; // Utiliza un renglón con peso 1

  gbc.gridwidth = GridBagConstraints.REMAINDER;
  gbc.anchor = GridBagConstraints.NORTH;
  gbag.setConstraints(heading, gbc);

  // Alinea los componentes a la derecha
  gbc.anchor = GridBagConstraints.EAST;

  gbc.gridwidth = GridBagConstraints.RELATIVE;
  gbag.setConstraints(regWDLab, gbc);
```

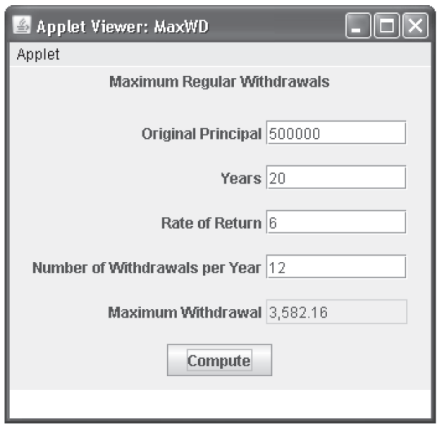
```
        gbc.gridwidth = GridBagConstraints.REMAINDER;
        gbag.setConstraints(regWDText, gbc);

        gbc.gridwidth = GridBagConstraints.RELATIVE;
        gbag.setConstraints(periodLab, gbc);
        gbc.gridwidth = GridBagConstraints.REMAINDER;
        gbag.setConstraints(periodText, gbc);

        gbc.gridwidth = GridBagConstraints.RELATIVE;
        gbag.setConstraints(rateLab, gbc);
        gbc.gridwidth = GridBagConstraints.REMAINDER;
        gbag.setConstraints(rateText, gbc);

        gbc.gridwidth = GridBagConstraints.RELATIVE;
        gbag.setConstraints(numWDLab, gbc);
        gbc.gridwidth = GridBagConstraints.REMAINDER;
        gbag.setConstraints(numWDText, gbc);

        gbc.gridwidth = GridBagConstraints.RELATIVE;
        gbag.setConstraints(initialLab, gbc);
        gbc.gridwidth = GridBagConstraints.REMAINDER;
        gbag.setConstraints(initialText, gbc);

        gbc.anchor = GridBagConstraints.CENTER;
        gbag.setConstraints(doIt, gbc);

        // Agrega todos los componentes.
        add(heading);
        add(regWDLab);
        add(regWDText);
        add(periodLab);
        add(periodText);
        add(rateLab);
        add(rateText);
        add(numWDLab);
        add(numWDText);
        add(initialLab);
        add(initialText);
        add(doIt);

        // Registra los listeners para recibir los eventos de acción
        regWDText.addActionListener(this);
        periodText.addActionListener(this);
        rateText.addActionListener(this);
        numWDText.addActionListener(this);
        doIt.addActionListener(this);

        // Crea un formato numérico
        nf = NumberFormat.getInstance();
        nf.setMinimumFractionDigits(2);
        nf.setMaximumFractionDigits(2);
    }

    / * El usuario presionó Enter en un campo de texto o presionó
        el botón Calcular. Muestra el resultado si todos los campos
        están completos*/
```

```
  public void actionPerformed(ActionEvent ae) {
      double result = 0.0;

    String regWDStr = regWDText.getText();
    String periodStr = periodText.getText();
    String rateStr = rateText.getText();
    String numWDStr = numWDText.getText();

    try {
      if(regWDStr.length() != 0 &&
        periodStr.length() != 0 &&
        rateStr.length() != 0 &&
        numWDStr.length() != 0) {

        regWDAmount = Double.parseDouble(regWDStr);
        numYears = Double.parseDouble(periodStr);
        rateOfRet = Double.parseDouble(rateStr) / 100;
        numPerYear = Integer.parseInt(numWDStr);

        result = compute();

        initialText.setText(nf.format(result));
      }

      showStatus(" "); // borra cualquier mensaje previo de error
    } catch (NumberFormatException exc) {
      showStatus("Datos Inválidos");
      initialText.setText("") ;
    }
  }

  // Calcula la inversion inicial requerida.
  double compute () {
    double b, e;
    double t1, t2;

    t1 = (regWDAmount * numPerYear) / rateOfRet;

    b = (1 + rateOfRet/numPerYear) ;
    e = numPerYear * numYears;

    t2 = 1 - (1 / Math.pow(b, e));

    return t1 * t2;
  }
}
```

Calcular la anualidad máxima para una inversión dada

Otro cálculo de anualidad calcula la máxima anualidad (en términos de un retiro regular) disponible desde una inversión dada sobre un periodo de tiempo especificado. Por ejemplo, si se tienen $500 000 en una cuenta de retiro, ¿qué cantidad se puede retirar cada mes durante 20 años, asumiendo un 6 por ciento de tasa de retorno? La fórmula que calcula el máximo retiro se muestra a continuación:

FIGURA 32-5
El applet **MaxWD**



$$\text{Retiro Máximo} = principal * (((rateOfRet / wdPerYear) / (-1 + ((rateOfRet / wdPerYear) + 1)^{wdPerYear*numYears})) + (rateOfRet / wdPerYear))$$

Donde *rateOfRef* especifica la tasa de devolución, *principal* contiene el valor de la inversión inicial, *wdPerYear* especifica el número de retiros por año y *numYears* especifica la duración de la renta en años.

El applet **MaxWD** mostrado a continuación calcula el monto máximo de los retiros regulares que se pueden hacer sobre un periodo de tiempo dado para una tasa de devolución dada. El applet producido por este programa se muestra en la Figura 32-5.

```
/* Calcula la renta máxima que puede
   retirarse de una inversión en un
   periodo de tiempo dado */
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.text.*;
/*
  <applet code="MaxWD" width=340 height=260>
  </applet>
*/

public class MaxWD extends JApplet
  implements ActionListener {

  JTextField maxWDText, orgPText, periodText,
             rateText, numWDText;
  JButton doIt;

  double principal; // monto inicial
  double rateOfRet; // taza de retorno anual
  double numYears; // tiempo en años
```

```
  int numPerYear; // cantidad de retiros por año

  NumberFormat nf;
  public void init() {
    try {
      SwingUtilities.invokeAndWait(new Runnable() {
        public void run() {
          makeGUI(); // inicializa la interfaz gráfica
        }
      });
    } catch(Exception exc) {
      System.out.println("No se puede crear la aplicación debido a: "+ exc);
    }
  }

  // crea e inicializa la interfaz gráfica
  private void makeGUI() {

// Utiliza un GridBagLayout
    GridBagLayout gbag = new GridBagLayout();
    GridBagConstraints gbc = new GridBagconstraints();
    setLayout(gbag);

    JLabel heading = new
           JLabel("Cantidad máxima de retiros regulares");

    JLabel orgPLab = new JLabel("Inversión original ");
    JLabel periodLab = new JLabel ("Años");
    JLabel rateLab = new JLabel("Tasa de retorno ");
    JLabel numWDLab =
           new JLabel ("Número de retiros por año ");
    JLabel maxWDLab = new JLabel("Número Máximo de retiros ");

    maxWDText = new JTextField(10);
    periodText = new JTextField(10);
    orgPText = new JTextField(10);
    rateText = new JTextField(10);
    numWDText = new JTextField(10);

    // Campo para mostrar el monto máximo de los de retiros
    maxWDText.setEditable(false);

    doIt = new JButton("Calcular");

    // Define el GridBagLayout
    gbc.weighty = 1.0; // utiliza un renglón de peso 1

    gbc.gridwidth = GridBagConstraints.REMAINDER;
    gbc.anchor = GridBagConstraints.NORTH;
    gbag.setConstraints(heading, gbc);

    // Alinea los componentes a la derecha
    gbc.anchor = GridBagConstraints.EAST;

    gbc.gridwidth = GridBagConstraints.RELATIVE;
    gbag.setConstraints(orgPLab, gbc);
    gbc.gridwidth = GridBagConstraints.REMAINDER;
    gbag.setConstraints(orgPText, gbc);
```

```
    gbc.gridwidth = GridBagConstraints.RELATIVE;
    gbag.setConstraints(periodLab, gbc);
    gbc.gridwidth = GridBagConstraints.REMAINDER;
    gbag. setConstraints (periodText, gbc);

    gbc.gridwidth = GridBagConstraints.RELATIVE¡
    gbag.setConstraints(rateLab, gbc);
    gbc.gridwidth = GridBagConstraints.REMAINDER;
    gbag.setConstraints(rateText, gbc);

    gbc.gridwidth = GridBagConstraints.RELATIVE;
    gbag.setConstraints(numWDLab, gbc);
    gbc.gridwidth = GridBagConstraints.REMAINDER;
    gbag.setConstraints(numWDText, gbc);

    gbc.gridwidth = GridBagConstraints.RELATIVE;
    gbag.setConstraints(maxWDLab, gbc);
    gbc.gridwidth = GridBagConstraints.REMAINDER;
    gbag.setConstraints(maxWDText, gbc);

    gbc.anchor = GridBagConstraints.CENTER;
    gbag.setConstraints(doIt, gbc);

    // Agrega todos los componentes.
    add(heading);
    add(orgPLab);
    add(orgPText);
    add(periodLab);
    add(periodText);
    add(rateLab);
    add(rateText);
    add(numWDLab);
    add(numWDText);
    add(maxWDLab);
    add(maxWDText);
    add(doIt);

    // Registra los listeners para recibir los eventos de acción
    orgPText.addActionListener(this);
    periodText.addActionListener(this);
    rateText.addActionListener(this);
    numWDText.addActionListener (this);
    doIt.addActionListener(this);

    // Crea un formato numérico
    nf = NumberFormat.getInstance();
    nf.setMinimumFractionDigits(2);
    nf.setMaximumFractionDigits(2);
  }

  /* El usuario presionó Enter en un campo de texto
     o presionó el botón Calcular. Muestra el resultado
     si todos los campos están completos. */
  public void actionPerformed(ActionEvent ae) {
    double result = 0.0;
```

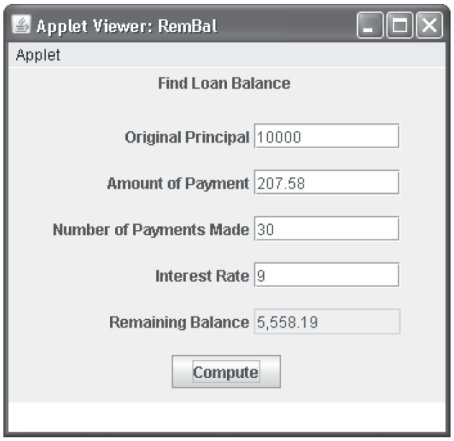
```
    String orgPStr = orgPText.getText();
    String periodStr = periodText.getText();
    String rateStr = rateText.getText();
    String numWDStr = numWDText.getText();

    try {
      if(orgPStr.length() != 0 &&
        periodStr.length() != 0 &&
        rateStr.length() != 0 &&
        numWDStr.length() != 0) {

        principal = Double.parseDouble(orgPStr);
        numYears = Double.parseDouble(periodStr);
        rateOfRet = Double.parseDouble(rateStr) / 100;
        numPerYear = Integer.parseInt(numWDStr);

        result = compute();

        maxWDText.setText(nf.format(result));
      }

      showStatus(""); // borra cualquier mensaje de error previo
    } catch (NumberFormatException exc) {
      showStatus("Datos inválidos");
      maxWDText.setText("");
    }
  }

  // Calcula los retiros regulares máximos
  double compute() {
    double b, e;
    double tl, t2;

    tl = rateOfRet / numperYear;

    b = (1 + tl);
    e = numPerYear * numYears;

    t2 = Math.pow(b, e) - 1;

    return principal * (tl/t2 + tl);
  }
}
```


Calcular el balance restante de un préstamo

Frecuentemente, se querrá conocer el balance restante de un préstamo. Esto es muy fácil de calcular si se sabe el préstamo original, la tasa de interés, la duración del préstamo y el número de pagos realizados. Para encontrar el balance restante, se deben sumar los pagos, restar de cada pago la cantidad de intereses y entonces restar el resultado al préstamo original.

El applet **RemBal**, listado a continuación, muestra cómo se calcula el balance restante de un préstamo. El applet producido por este programa se muestra en la Figura 32-6.

FIGURA 32-6
El applet **RemBal**




```
// Encuentra el balance restante un préstamo
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.text.*;
/*
  <applet code="RemBal" width=340 height=260>
  </applet>
*/

public class RemBal extends JApplet
  implements ActionListener {

  JTextField orgPText, paymentText, remBalText,
             rateText, numpayText;
  JButton doIt;

  double orgprincipal; // préstamo original
  double intRate; // tasa de interés
  double payment; // monto de cada pago
  double numpayments; // número de pagos hechos

  /* Número de pagos por año. Se podría hacer que
  el usuario pudiera editar este valor */
  final int payPerYear = 12;

  NumberFormat nf;

  public void init() {
    try {
      SwingUtilities.invokeAndWait(new Runnable () {
        public void run() {
          makeGUI(); // inicializa la interfaz gráfica
        }
      });
```

```
  } catch(Exception exc) {
    System.out.println("No se puede crear la aplicación debido a" + exc);
  }
}

// Crea e inicializa la interfaz gráfica
private void makeGUI() {

  // Utiliza un GridBagLayout.
  GridBagLayout gbag = new GridBagLayout();
  GridBagConstraints gbc = new GridBagConstraints();
  setLayout(gbag);

  JLabel heading = new
         JLabel("Encuentra el Balance Restante de un Préstamo "),

  JLabel orgPLab = new JLabel("Préstamo original "),
  JLabel paymentLab = new JLabel ("Cantidad pagada "),
  JLabel numPayLab = new JLabel("Número de pagos realizados "),
  JLabel rateLab = new JLabel("Tasa de Interés "),
  JLabel remBalLab = new JLabel("Balance restante"),

  orgPText = new JTextField(10);
  paymentText = new JTextField(10);
  remBalText = new JTextField(10);
  rateText = new JTextField(10);
  numPayText = new JTextField(10);

  // campo de pagos
  remBalText. setEditable (false);

  doIt = new JButton("Calcular");

  // Define el GridBagLayout.
  gbc.weighty = 1.0; // utiliza un renglón con peso 1

  gbc.gridwidth = GridBagConstraints.REMAINDER;
  gbc.anchor = GridBagConstraints.NORTH;
  gbag.setConstraints(heading, gbc);

  // Alinea los componentes a la derecha
  gbc.anchor = GridBagConstraints.EAST;

  gbc.gridwidth = GridBagConstraints.RELATIVE;
  gbag.setConstraints(orgPLab, gbc);
  gbc.gridwidth = GridBagConstraints.REMAINDER;
  gbag.setConstraints(orgPText, gbc);

  gbc.gridwidth = GridBagConstraints.RELATIVE;
  gbag.setConstraints(paymentLab, gbc);
  gbc.gridwidth = GridBagConstraints.REMAINDER;
  gbag.setConstraints(paymentText, gbc);

  gbc.gridwidth = GridBagConstraints.RELATIVE;
  gbag.setConstraints(rateLab, gbc);
```

```
    gbc.gridwidth = GridBagConstraints.REMAINDER;
    gbag.setConstraints(rateText, gbc);

    gbc.gridwidth = GridBagConstraints.RELATIVE;
    gbag.setConstraints(numpayLab, gbc);
    gbc.gridwidth = GridBagConstraints.REMAINDER;
    gbag.setConstraints(numPayText, gbc);

    gbc.gridwidth = GridBagConstraints.RELATIVE;
    gbag.setConstraints(remBalLab, gbc);
    gbc.gridwidth = GridBagConstraints.REMAINDER;
    gbag.setConstraints(remBalText, gbc);

    gbc.anchor = GridBagConstraints.CENTER;
    gbag.setConstraints(doIt, gbc);

    // Agrega todos los componentes.
    add(heading);
    add(orgPLab);
    add(orgPText);
    add(paymentLab);
    add(paymentText);
    add(numPayLab);
    add(numPayText);
    add(rateLab);
    add(rateText);
    add(remBalLab);
    add(remBalText);
    add(doIt);

    // Registra los listeners para recibir los eventos de acción
    orgPText.addActionListener(this);
    numPayText.addActionListener(this);
    rateText.addActionListener(this);
    paymentText.addActionListener(this);
    doIt.addActionListener(this);

    // Crea un formato numérico
    nf = NumberFormat.getInstance();
    nf.setMinimumFractionDigits(2);
    nf.setMaximumFractionDigits(2);
  }

  /* El usuario presionó Enter en un campo de texto
     o presionó el botón Calcular. Muestra el resultado
     si todos los campos están completos*/
  public void actionPerformed(ActionEvent ae) {
    double result = 0.0;

    String orgPStr = orgPText.getText();
    String numpayStr = numPayText.getText();
    String rateStr = rateText.getText();
```

```
    String payStr = paymentText.getText();

    try {
      if(orgPStr.length() != 0 &&
          numPayStr.length() != 0 &&
          rateStr.length() != 0 &&
          payStr.length() != 0) {

        orgPrincipal = Double.parseDouble(orgPStr);
        numpayments = Double.parseDouble(numPayStr);
        intRate = Double.parseDouble(rateStr) / 100;
        payment = Double.parseDouble(payStr);

        result = compute();

        remBalText.setText(nf.format(result));
      }

      showStatus(""); // borrar cualquier mensaje de error anterior
    } catch (NumberFormatException exc) {
      showstatus("Datos Inválidos");
      remBalText.setText("") ;
    }
  }

  // Calcula el balance del préstamo.
  double compute() {
    double bal = orgPrincipal;
    double rate = intRate / payPerYear;

    for(int i = 0; i < numpayments; i++)
      bal -= payment - (bal * rate);

    return bal;
  }
}
```


Crear servlets financieros

Aunque los applets son fáciles de crear y de usar, éstos son sólo la mitad de la ecuación de Java para Internet. La otra mitad son los servlets. Los servlets se ejecutan del lado del servidor y son más apropiados para algunas aplicaciones. Debido a que muchos de los lectores podrían querer utilizar servlets en lugar de applets en sus aplicaciones comerciales, en lo que resta del capítulo mostraremos cómo convertir los applets financieros en servlets.

Debido a que todos los applets financieros utilizan la misma estructura básica, revisaremos la conversación de sólo un applet: **RegPay**. El mismo proceso básico se puede aplicar para convertir cualquier applet en un servlet. Como veremos, éste no es un proceso demasiado complicado.

NOTA *Para mayor información sobre creación, prueba y ejecución de servlets, véase el Capítulo 31.*

Convertir un applet en un servlet

Es sencillo convertir el applet **RegPay** en un servlet. Primero, el servlet debe importar los paquetes **javax.servlet** y **java.servlet.http**. También debe extender de **HttpServlet**, no de **JApplet**. A continuación, se debe eliminar todo el código de la interfaz gráfica. Luego, debe agregar el código que obtiene los parámetros que se pasan al servlet mediante la página HTML que llama al servlet. Finalmente, el servlet debe enviar al cliente el código HTML que mostrará los resultados. Los cálculos financieros básicos permanecen iguales. La única diferencia es la forma en que se obtienen los datos y la forma en que muestra el resultado.

El servlet RegPayS

La siguiente clase, llamada **RegPayS**, es la versión en servlet del applet **RegPay**. El código está escrito asumiendo que el archivo **RegPayS.class** será almacenado en el directorio de donde radican los servlets de ejemplo en Tomcat, como se describió en el Capítulo 31. Recuerde modificar el archivo **web.xml** como se describió en el Capítulo 31.

```
// Un servlet simple que calcula los pagos de un préstamo.
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.text.*;

public class RegpayS extends HttpServlet {
  double principal; // préstamo original
  double intRate; // tasa de interés
  double numYears; // duración del préstamo en años

  /* Número de pagos por año. Se podría permitir al usuario
  editar este valor. */
  final int payPerYear = 12;

  NumberFormat nf;

  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
    throws ServletException, IOException {
    String payStr = " ";

    // Crea un formato numérico
    nf = NumberFormat.getInstance();
    nf.setMinimumFractionDigits(2);
    nf.setMaximumFractionDigits(2);

    // Obtiene los parámetros.
    String amountStr = request.getParameter("cantidad");
    String periodStr = request.getParameter("periodo");
    String rateStr = request.getParameter("tasa");

    try {
      if(amountStr != null && periodStr != null &&
         rateStr != null) {
        principal = Double.parseDouble(amountStr);
```

```
      numYears = Double.parseDouble(periodStr);
      intRate = Double.parseDouble(rateStr) / 100;

      payStr = nf.format(compute());
    } else { // faltan uno o más parámetros
      amountStr = "";
      periodStr = "";
      rateStr = "";
    }
  } catch (NumberFormatException exc) {
    // Toma las acciones apropiadas.
  }

  // Define el tipo de contenido
  response.setContentType("text/html") ;

  // Obtiene el flujo de salida
  PrintWriter pw = response.getWriter();

  // Muestra el HTML necesario.
  pw.print("<html><body> <left>" +
           "<form name=\"Form1\"" +
           " action=\"http://localhost:8080/" +
           "servlets-examples/servlet/RegPayS\">" +
           "<B>Ingrese el monto a financiar:</B>" +
           " <input type=textbox name=\"cantidad\"" +
           " size=12 value=\"");
  pw.print(amountStr + "\">");
  pw.print("<BR><B>Ingrese la duración en años:</B>" +
           " <input type=textbox name=\"periodo\""+
           " size=12 value=\"");
  pw.println(periodStr + "\">");
  pw.print("<BR><B>Ingrese la tasa de interés:</B>" +
           " <input type=textbox name=\"tasa\"" +
           " size=12 value=\"");
  pw.print(rateStr + "\">");
  pw.print("<BR><B>Pagos mensuales:</B>" +
           " <input READONLY type=textbox" +
           " name=\"payment\" size=12 value=\"");
  pw.print(payStr + "\">");
  pw.print("<BR><P><input type=submit value=\"Enviar\">");
  pw.println("</form> </body> </html>");
}

// Calcula el préstamo
double compute() {
  double numer;
  double denom;
  double b, e;

  numer = intRate * principal / payPerYear;
```

```
    e = -(payPerYear * numYears);
    b = (intRate / payPerYear) + 1.0;

    denom = 1.0 - Math.pow(b, e);

    return numer / denom;
  }
}
```

Lo primero que se debe notar sobre **RegPayS** es que tiene sólo dos métodos: **doGet( )** y **compute( )**. El método **compute( )** es el mismo que utilizó el applet. El método **doGet( )** está definido por la clase **HttpServlet**, de la cual extiende de **RegPayS**. Este método es llamado por el servidor cuando el servlet debe responder a la solicitud GET. Note que se pasa como referencia a los objetos **HttpServletRequest** y **HttpServletResponse** asociados con la petición.

Para el parámetro solicitado, el servlet obtiene los argumentos asociados con la solicitud. Esto se hace llamando al método **getParameter( )**. Los parámetros se regresan en forma de cadena. De esta forma, un valor numérico debe ser manualmente convertido a un formato binario. Si no hay parámetros disponibles, se devuelve un valor **null**.

Para el objeto **response**, el servlet obtiene un flujo en el cuál se puede escribir la información que se enviará al cliente. Antes de obtener un objeto **PrintWriter** para el flujo de respuesta, el tipo de salida debería ser definido para text/html llamando al método **setContentType( )**.

RegPayS puede ser llamado con o sin parámetros. Si se llama a **RegPayS** sin parámetros, el servlet responde con el código HTML necesario para mostrar una forma vacía. En caso contrario, cuando es llamado con todos los parámetros necesarios, **RegPayS** calcula los pagos del préstamo y muestra de nuevo la forma, mostrando el valor correspondiente en el campo de pago. En la Figura 32-7 se muestra el servlet **RegPayS** en acción.

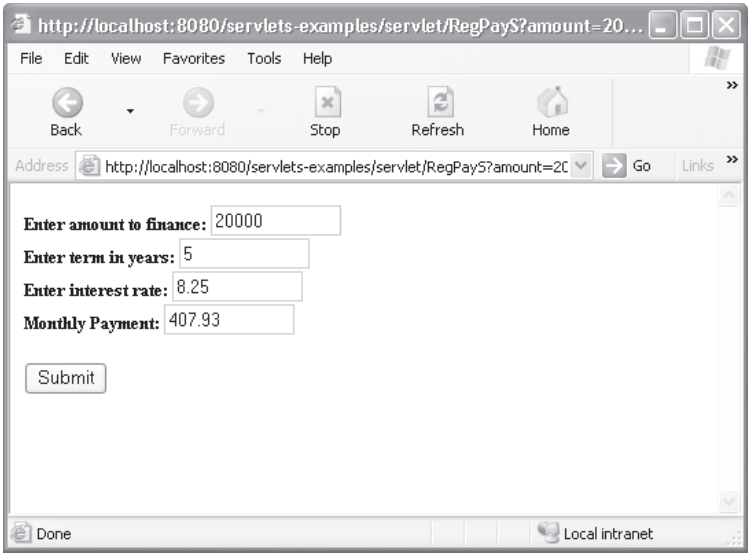


FIGURA 32-7
El servlet **RegPays** en acción

La forma más simple de invocar a **RegPayS** es enlazarnos a su URL sin pasarle ningún parámetro. Por ejemplo, asumiendo que se está utilizando Tomcat, se puede utilizar la siguiente línea para ejecutar al servlet:

```
< A HREF = "http://localhost:8080/servlets-examples/servlet/RegPayS">
Calculadora de Préstamos </A>
```

Esto muestra una liga llamada Calculadora de Préstamos, la cual nos enlaza con el servlet **RegPays** en el directorio de los servlets del ejemplo de Tomcat. Observe que no se pasa ningún parámetro. Esto hace que **RegPayS** devuelva el HTML que muestra una calculadora de préstamos vacía.

También se puede invocar **RegPayS** mostrando una forma vacía construida manualmente. Esta estrategia se muestra a continuación, una vez más utilizando el directorio de ejemplos de Tomcat.

```
<html>
<body>
<form name="Form1"
  action="http://localhost:8080/servlets-examples/servlet/RegPayS">
<B> Ingrese el monto a financiar:</B>
<input type=textbox name="cantidad" size=12 value= "">
<BR>
<B> Ingrese la duración en años:</B>
<input type=textbox name="periodo" size=12 value= "">
<BR>
<B> Ingrese la tasa de interés:</B>
<input type=textbox name="tasa" size=12 value= "">
<BR>
<B> Pagos mensuales:</B>
<input READONLY type=textbox name="pagos"
  size=12 value= "">
<BR><P>
<input type=submit value="Enviar">
</form>
</body>
</html>
```


Ejercicios recomendados

Lo primero que quizás quiera intentar es cambiar los otros applets financieros en servlets. Dado que todos los applets financieros son creados sobre el mismo esquema, simplemente se deben seguir los mismos pasos utilizados para **RegPayS**. Existen muchos otros cálculos financieros que posiblemente encuentre útiles para ser implementados como applets o servlets, tales como, la tasa de devolución de una inversión o el monto regular de depósito necesario sobre el tiempo para alcanzar un valor futuro. También podría generar la gráfica de la amortización de un préstamo. Otro buen ejercicio es intentar crear una gran aplicación que ofrezca todos los cálculos presentados en este capítulo, permitiendo a los usuarios seleccionar los cálculos deseados desde un menú.

CAPÍTULO 33

Creando un administrador de descargas en Java

Probablemente, alguna vez se le ha interrumpido la descarga de información desde Internet, con la consiguiente pérdida del avance logrado hasta el momento en que la conexión se interrumpió. Si nos conectamos a Internet con una conexión telefónica es probable que hayamos pasado por esta molestia. Desde la desconexión de la llamada, hasta la falla de la computadora puede dejar la descarga de un archivo incompleta. Para decir lo menos, reiniciar desde cero la descarga de la información una y otra vez puede consumir demasiado tiempo y ser una experiencia frustrante.

Un factor que en ocasiones se pasa por alto es que muchas descargas interrumpidas pudieran ser continuadas. Esto permite recomenzar la descarga desde el punto en el cual se interrumpió, en lugar de tener que comenzar de nuevo. En este capítulo se desarrolla una herramienta llamada Download Manager la cual permite administrar descargas de información de Internet y simplifica el trabajo de continuar descargas interrumpidas. Esta herramienta además, permite detener y continuar una descarga, así como administrar múltiples descargas simultáneamente.

Una de las características más importantes del Download Manager es su habilidad para descargar sólo una porción específica de un archivo. En un escenario clásico de descarga se descarga de principio a fin un archivo completo. Si la transmisión del archivo es interrumpida, por cualquier razón, los datos descargados hasta ese momento se pierden. El Download Manager, puede continuar donde la interrupción ocurrió y descargar la parte faltante del archivo. Sin embargo, no todas las descargas son iguales, sin embargo, algunas no pueden ser continuadas. Los detalles de cuales archivos pueden o no ser descargados en partes, se explican en la siguiente sección.

El Download Manager no sólo es una herramienta útil, es un excelente ejemplo del poder y precisión de las API incluidas en Java, especialmente cuando se utilizan para crear interfaces con Internet. Debido a que Internet fue la fuerza que impulsó la creación de Java, no es de sorprender que las capacidades para trabajo en red de Java sean insuperables. Por ejemplo, intentar crear la herramienta Download Manager en otro lenguaje, como C++, implicaría sin duda, mayores problemas y esfuerzo.

Introducción

Para entender y apreciar el trabajo realizado por el Download Manager es necesario comprender la forma en que realmente opera la descarga de información desde Internet.

Las descargas de información vía Internet en su forma más simple son simplemente transacciones cliente/servidor. El cliente, nuestro navegador, solicita la descarga de un archivo localizado en un servidor en Internet. Luego, el servidor responde enviando el archivo solicitado al navegador. Para que los clientes se comuniquen con los servidores debe existir un protocolo de comunicación. Los protocolos comúnmente utilizados para descargar archivos son: File Transfer Protocol (FTP) y Hyper Text Transfer Protocol (HTTP). FTP es usualmente asociado de forma genérica con el intercambio de archivos entre computadoras mientras que HTTP es usualmente asociado específicamente con transferencia de páginas Web y sus archivos relacionados (esto es, gráficos, sonidos, etc.). Con el tiempo, conforme el Web creció en popularidad, HTTP se convirtió en el protocolo dominante para descargar archivos desde Internet. Sin embargo, FTP no está extinto.

A fin de ser breves, el Download Manager desarrollado en este capítulo sólo trabajará con el protocolo HTTP. No obstante, agregar soporte para el protocolo FTP sería un excelente ejercicio para extender el programa. Las descargas en HTTP se realizan en dos formas: reiniciable (HTTP 1.1) y no reiniciable (HTTP 1.0). La diferencia entre estas dos formas recae en la forma en que los archivos pueden ser solicitados a los servidores. Con el antiguo protocolo HTTP 1.0, un cliente sólo podía solicitar al servidor el envío del archivo, mientras que con HTTP 1.1, un cliente puede solicitar al servidor el envío del archivo completo o el envío de una *porción específica del archivo*. Ésta será la característica sobre la cuál estará construido el Download Manager.

Descripción del administrador de descargas

El Download Manager utiliza una simple pero efectiva interfaz gráfica construida con las bibliotecas de Swing. La ventana del Download Manager se muestra en la Figura 33-1. El uso de Swing da a la interfaz una apariencia moderna y viva.

La interfaz gráfica mantiene la lista de las descargas en ejecución. Cada descarga en la lista reporta su URL, tamaño del archivo en bytes, el porcentaje de avance, y su estatus actual. Las descargas pueden estar en uno de los siguientes estatus: DOWNLOADING, PAUSED, ERROR, COMPLETE o CANCELLED. La interfaz gráfica además tiene controles para añadir descargas a la lista y para cambiar el estado de cada descarga de la lista. Cuando una descarga en la lista es seleccionada, dependiendo de su estado actual, ésta puede ser puesta en pausa, reiniciada, cancelada, o eliminada de la lista.

El Download Manager está separado en unas cuantas clases, siguiendo la separación natural de la funcionalidad de cada componente. Estas clases son **Download**, **DownloadsTableModel**, **ProgressRender**, y **DownloadManager**. La clase **DownloadManager** es responsable de la interfaz gráfica y hace uso de las clases **DownloadsTableModel** y **ProgressRender** para desplegar la lista actual de descargas. La clase **Download** representa una descarga "administrada"y es responsable de realizar la descarga del archivo. En la siguiente sección, revisaremos cada una de estas clases a detalle, resaltando su funcionamiento interno y explicando como se relacionan entre ellas.

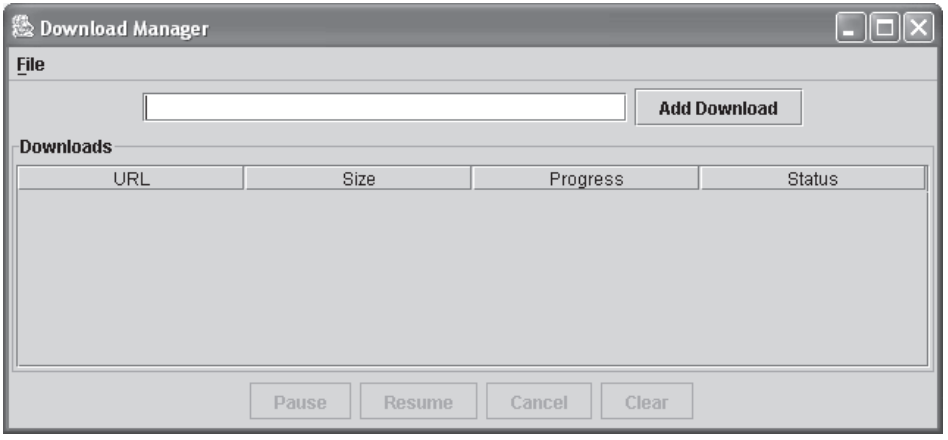


FIGURA 33-1 La interfaz gráfica de la aplicación Download Manager

La clase Download

La clase **Download** es el centro de la aplicación Download Manager. Su propósito principal es descargar un archivo y guardarlo en el disco. Cada vez que una nueva descarga se añade al Download Manager, un nuevo objeto de tipo **Download** es instanciado para gestionar la descarga.

El Download Manager tiene la habilidad de descargar múltiples archivos a la vez. Para lograr esto, es necesario que cada una de las descargas simultáneas sea ejecutada independientemente. Además es necesario que cada descarga administre su propio estado de forma que pueda ser reflejado en la interfaz gráfica. Ésta es la labor de la clase **Download**.

El código completo de la clase **Download** se muestra a continuación. Nótese que la clase hereda de la clase **Observable** e implementa la interfaz **Runnable**. Cada una de sus partes se examina con detalle en las secciones siguientes.

```
import java.io.*;
import java.net.*;
import java.util.*;

// Esta clase descarga un archivo desde una dirección URL.
class Download extends Observable implements Runnable {
  // tamaño máximo del bufer de descarga
  private static final int MAX_BUFFER_SIZE = 1024;

  // Estos son los nombres para los estados posibles
  public static final String STATUSES[] = {"Downloading",
    "Paused", "Complete", "Cancelled", "Error"};

  // Estos son los códigos de estado
  public static final int DOWNLOADING = 0;
  public static final int PAUSED = 1;
  public static final int COMPLETE = 2;
  public static final int CANCELLED = 3;
  public static final int ERROR = 4;
```

```
private URL url; // URL donde se localiza el archivo
private int size; // tamaño del archivo en bytes
private int downloaded; // número de bytes descargados
private int status; // estado actual de la descarga

// Constructor de la clase Download.
public Download(URL url) {
  this.url = url;
  size = -1;
  downloaded = 0;
  status = DOWNLOADING;

  // Comenzar la descarga
  download() ;
}

// Obtener el valor del campo URL
public String getUrl() {
  return url.toString();
}

// Obtener el valor del campo size
public int getSize() {
return size;
}

// Obtener el valor del campo progress
public float getProgress() {
  return ((float) downloaded / size) * 100;
}

// Obtener el valor del campo status.
public int getStatus() {
  return status;
}

// Generar una pausa en la descarga
public void pause() {
  status = PAUSED;
  stateChanged();
}

// Reiniciar la descarga del archivo
public void resume() {
  status = DOWNLOADING;
  stateChanged();
  download();
}

// Cancelar la descarga
public void cancel() {
  status = CANCELLED;
  stateChanged();
}
```

```
// Indicar que la descarga actual tiene problemas
private void error() {
  status = ERROR;
  stateChanged();
}

// Iniciar o reiniciar una descarga
private void download() {
  Thread thread = new Thread(this);
  thread.start();
}

// Obtener el nombre del archivo a partir de su URL
private String getFileName(URL url)
  String fileName = url.getFile();
  return fileName.substring(fileName.lastlndexOf('/') + 1);
}

// Descargar el archivo
public void run() {
  RandomAccessFile file = null;
  InputStream stream = null;

  try {
    // Abrna conexión al URL
    HttpURLConnection connection
      (HttpURLConnection) url.openConnection();

  // Especificar la porción del archivo a descargar
  connection.setRequestProperty ("Range",
    "bytes=" + downloaded + "-");

  // Conexión con el servidor
  connection.connect();

  // Asegurarse de que el servidor responda con un código 200
  if (connection.getResponseCode() / 100 != 2) {
    error();
  }

  // Revisar que el tamaño del archivo sea válido
  int contentLength = connection.getContentLength();
  if (contentLength < 1) {
    error();
  }

  /* Establecer el tamaño para la descarga
     en caso de que no se haya establecido con anterioridad */
  if (size == -1) {
    size = contentLength;
    stateChanged();
  }
```

```
      // Abrir el archivo y colocar el apuntador de archivo al final del mismo
      file = new RandomAccessFile(getFileName(url), "rw");
      file.seek(downloaded);

      stream = connection.getInputStream();
      while (status == DOWNLOADING) {
        /* Definir el tamaño del bufer
           acorde con la porción de archivo que falta por descargar */
        byte buffer[];
        if (size - downloaded > MAX_BUFFER_SIZE) {
          buffer = new byte[MAX BUFFER SIZE];
        } else {
          buffer = new byte [size - downloaded];
        }

        // Leer datos del servidor en bufer
        int read = stream.read(buffer);
        if (read == -1)
          break;

        // Escribir el bufer en el archivo
        file.write(buffer, 0, read);
        downloaded += read;
        stateChanged();
      }

      /* Cambia el estado de la transferencia a COMPLETE */
      if (status == DOWNLOADING) {
        status = COMPLETE;
        stateChanged();
      }
    } catch (Exception e) {
      error() ;
    } finally {
      // Cerrar el archivo
      if (file != null)
        try {
          file.close();
        } catch (Exception e) {}
      }

      // Cerrar la conexión con el servidor
      if (stream != null) {
        try {
          stream.close();
        } catch (Exception e) {}
      }
    }
  }
```

```
  // Notifica a los observadores que ha cambiado el estado de esta descarga
  Private void stateChanged() {
    setChanged();
    notifyObservers();
  }
}
```

Las variables de Download

La clase **Download** comienza declarando diversas variables con los modificadores **static final**, estas variables especifican valores constantes utilizados por la clase. Luego se declaran cuatro variables de instancia. La variable **url** almacena la dirección URL del archivo que se está descargando; la variable **size** almacena el tamaño del archivo en bytes; la variable **downloaded** almacena el número de bytes que han sido descargados hasta el momento; y la variable **status** indica el estado actual del proceso de descarga.

El constructor Download

Al constructor de la clase **Download** se le pasa como argumento una referencia al objeto URL que encapsula la dirección del archivo a descargar. El parámetro con la dirección **URL** se almacena en la variable de instancia **url**. Al resto de las variables de instancia se les asignan los valores iniciales correspondientes. Finalmente se invoca al método **download( )**. Nótese que la variable **size** recibe un valor inicial de -1 para indicar la ausencia de un tamaño conocido.

El método download( )

El método **download( )** crea un nuevo objeto de tipo **Thread**, al cual le pasa como referencia el objeto **Download** que invoca. Como se mencionó antes, es necesario que cada descarga se ejecute independientemente de las otras. A fin de que la clase **Download** trabaje sola, ésta debe ser ejecutada en su propio hilo. Java cuenta con un excelente soporte para trabajar con hilos de forma fácil. Para trabajar con hilos, la clase **Download** simplemente implementa la interfaz **Runnable** y sobrescribe el método **run( )**. Después de que el método **download( )** crea una nueva instancia de la clase **Thread** llama al método **start( )** de la instancia. Invocar al método **start( )** del hilo causa que el método **run( )** de la clase **Download** sea ejecutado.

El método run( )

Cuando se ejecuta el método **run( )**, las descargas actuales bajan de prioridad debido a su tamaño e importancia. Examinemos detenidamente el método línea por línea. El método **run( )** comienza con las siguientes líneas:

```
RandomAccessFile file = null;
InputStream stream = null;

try {
  // Abrir una conexión al URL
  HttpURLConnection connection
    (HttpURLConnection) url.openConnection();
```

Primero, el método **run( )** inicializa las variables necesarias para crear un flujo de red desde donde se leerá el contenido a descargar y crea un archivo en donde se colocará el contenido descargado. A continuación se abre la conexión con el URL donde radica el contenido, realizando una llamada **url.openConnection( )**. Dado que sabemos que nuestra aplicación Download Manager sólo soporta descargas vía http la conexión se transforma a un tipo **HttpUrlConnection**. La conversión a **HttpUrlConnection** nos permite aprovechar las características especificas de conexiones basadas en el protocolo http, por ejemplo, nos permite utilizar el método **getResponseCode( )**. Observe que la invocación a **url.openConnection( )** no crea una conexión al servidor. Simplemente crea un nuevo objeto de tipo **URLConnection** asociado con una dirección URL.

Después de que el objeto **HttpURLConnection** ha sido creado, las propiedades de la conexión se definen llamando al método **connection.setRequestProperty( )** como se muestra a continuación:

```
// Especificar la porción del archivo a descargar
connection.setRequestProperty ("Range",
  "bytes=" + downloaded + "-");
```

Establecer las propiedades de la descarga permite enviar información extra en la petición que se envía al servidor desde donde se descargará el archivo. En este caso se agrega la propiedad"Range". Esta propiedad es muy importante debido a que especifica el rango de los bytes que están siendo solicitados al servidor para descargarse. Normalmente, todos los bytes de un archivo son descargados en una sola operación. Sin embargo, si una descarga ha sido interrumpida o pausada, sólo se requiere descargar los bytes faltantes. Definir la propiedad "Range"es la base para el funcionamiento de nuestra aplicación Download Manager.

La propiedad"Range"se especifica de la siguiente forma:

start-byte – end-byte

Por ejemplo 0 – 12345". Sin embargo, el segundo valor es opcional. Si el byte final no se anota, el rango termina al final del archivo. El método **run( )** nunca especifica el byte final debido a que la descarga debe ejecutarse hasta que el archivo completo sea descargado a menos que sea interrumpido o colocado en pausa.

Las siguientes líneas en el método son éstas:

```
// Conexión con el servidor
connection.connect();

// Asegurarse de que el servidor responda con un código 200
íf (connection.getResponseCode() / 100 != 2) {
  error();
}

// Revisar que el tamaño del archivo sea válido
int contentLength = connection.getContentLength();
íf (contentLength < 1) {
  error();
}
```

El método **connection.connect( )** es llamado para establecer una conexión con el servidor. Luego, el código de respuesta entregado por el servidor es revisado. El protocolo http tiene una lista de códigos de respuesta que representan respuestas del servidor a una petición. Los

códigos de respuesta de http están organizados en rangos numéricos de 100 en 100. Los valores en el rango 200 indican éxito en la conexión. El código de respuesta entregado por el servidor se valida para garantizar que está en el rango 200 llamando a **connection.getResponseCode( )** y dividiendo entre 100. Si el valor de la división es 2, la conexión fue exitosa.

A continuación, el método **run( )** obtiene el tamaño del archivo a transferir llamando a **connection.getContentLength( )**. El tamaño representa el número de bytes en el archivo localizado en el servidor. Si el tamaño es menor a 1, se llama al método **error( )**. El método **error( )** actualiza el estado de la descarga al estado de **ERROR**, y luego llama al método **stateChanged( )**. El método **stateChanged( )** será descrito con detalle más adelante.

Después de obtener el tamaño, el siguiente código válida si ese valor está almacenado en la variable **size**:

```
/* Establecer el tamaño para la descarga
   en caso de que no se haya establecido con anterioridad */
if (size == -1) {
  size = contentLength;
  stateChanged();
}
```

Como se puede ver, en lugar de asignar la variable **contentLength** a la variable **size** directamente, sólo se hace esta asignación si nunca antes se había asignado un valor a **size**. La razón de esto es que la variable **contentLength** refleja la cantidad de bytes que el servidor enviará y no el tamaño completo del archivo. Esto es, el valor de la variable **contentLength** depende del rango especificado para la descarga, sólo cuando el rango es (0-), el valor de **contentLength** es el tamaño del archivo.

Las siguientes líneas de código mostradas a continuación crean un objeto de tipo **RandomAccessFile** utilizando el mismo nombre de archivo definido en el objeto URL.

```
// Abrir el archivo y colocar el apuntador de archivo al final del mismo
file = new RandomAccessFile(getFileName(url), "rw");
file.seek(downloaded);
```

El objeto **RandomAccessFile** se abre en modo"rw", lo cual especifica que el archivo puede ser escrito y leído. Una vez que el archivo es abierto, el método **run( )** busca el final del archivo llamando al método **file.seek( )**, al cual se le pasa como argumento la variable downloaded. Esto coloca el apuntador del archivo al final del mismo. Es necesario colocar el apuntador del archivo al final en caso de que se esté reiniciando una descarga para que los bytes que llegan no sobrescriban a los existentes. Una vez listo el archivo, se obtiene un flujo de lectura a partir de la conexión con el servidor utilizando el método **connection.getInputStream( )**, como se muestra a continuación:

```
stream = connection.getInputStream();
```

El corazón de toda la acción comienza a continuación con el siguiente ciclo **while**:

```
while (status == DOWNLOADING) {
  /* Definir el tamaño del bufer
     acorde con la cantidad de archivo que falta por descargar */
  byte buffer[];
  if (size - downloaded > MAX_BUFFER_SIZE) {
    buffer = new byte[MAX BUFFER SIZE];
```

```
} else {
buffer = new byte [size - downloaded];
}

// Leer datos del servidor en bufer
int read = stream.read(buffer);
if (read == -1)
break;

// Escribir el bufer en el archivo
file.write(buffer, 0, read);
downloaded += read;
stateChanged();
}
```

Este ciclo continúa su ejecución hasta que la variable **status** cambia al estado de **DOWNLOADING**. Dentro del ciclo, se crea un arreglo de valores de tipo byte para almacenar los bytes que serán descargados. El tamaño del arreglo se establece acorde a la cantidad de bytes que faltan por ser descargados. Si la cantidad de bytes que faltan por descargar es mayor que el valor de **MAX_BUFFER_SIZE**, se utiliza **MAX_BUFFER_SIZE** como tamaño del bufer. En otro caso, el bufer tendrá un tamaño igual a la cantidad de bytes que faltan por descargar. Una vez que el bufer tiene un tamaño, la descarga comienza con la llamada al método **stream.read( )**. Este método lee bytes del servidor y los coloca en el bufer. El método regresa la cantidad de bytes que leyó. Si el número de bytes leídos es igual a –1 la descarga ha sido completada y el ciclo termina. En caso contrario, la descarga no termina y los bytes que han sido leídos se escriben en el disco llamando al método **file.write( )**. Luego la variable **downloaded** se actualiza para reflejar el número de bytes descargados hasta el momento. Finalmente, dentro del ciclo se invoca al método **stateChanged( )**.

Después de que termina la ejecución del ciclo, las siguientes líneas de código revisan el motivo de la terminación del ciclo:

```
/* Cambia el estado de la transferencia a COMPLETE */
if (status == DOWNLOADING) {
  status = COMPLETE;
  stateChanged();
}
```

Si el estado de la descarga aún es **DOWNLOADING**, esto significa que el ciclo terminó porque la descarga se completó. En caso contrario, el ciclo terminó porque el estado de la descarga cambió a un valor diferente de **DOWNLOADING**.

El método **run( )** finaliza con los bloques **catch** y **finally** mostrados a continuación:

```
} catch (Exception e) {
  error();
} finally {
  // Cerrar el archivo
  if (file != null)
    try {
      file.close();
    } catch (Exception e) {}
  }
  // Cerrar la conexión con el servidor
  if (stream != null) {
    try {
      stream.close();
```

```
    } catch (Exception e) {}
  }
}
```

En caso de generarse una excepción durante el proceso de descarga, el bloque **catch** captura la excepción y llama al método **error( )**. El bloque **finally** se asegura de que las conexiones tanto del archivo, variable **file**, como de la conexión al servidor, variable **stream**, se cierren sin importar si ocurrió o no una excepción.

El método stateChanged( )

A fin de que la aplicación Download Manager despliegue información actualizada de cada una de las descargas que administra, debe tener información de cada cambio que ocurra en una descarga. Para ello se utiliza el patrón de diseño Observer. El patrón Observer es análogo a una lista de distribución de noticias donde mucha gente se registra para recibir comunicados. Cada vez que se genera un nuevo comunicado, todas las personas en la lista reciben un mensaje con ese comunicado. En el caso del patrón Observer, existe una clase observada a la cual las clases que la desean observar se registran para obtener notificaciones de los cambios que en ella ocurren.

La clase **Download** emplea el patrón Observer extendiendo la clase **Observable** de Java. Extender la clase Observable les permite a las clases que implementan la interfaz **Observer** registrarse con la clase **Download** para recibir notificaciones de cambios de estado. Cada vez que la clase **Download** requiere notificar a sus observadores sobre cambios de estado invoca al método **stateChanged( )**. El método **stateChanged( )** primero llama al método **setChanged( )** definido en la clase Observable para indicar que la clase ha sufrido cambios. Luego el método **setChanged( )** llama al método **notifyObservers( )** declarado en la clase Observable, el cual difunde la notificación de cambio a los observadores (objetos que implementaron la interfaz **Observer**) registrados.

Los métodos de acción y accesores

La clase **Download** tiene numerosos métodos de acción y accesores para controlar la descarga y obtener información de ésta. Los métodos de acción **pause( )**, **resume( )** y **cancel( )** realizan la actividad que su nombre (en inglés) indica: coloca la descarga en pausa, la reinicia o la cancela, respectivamente. Similarmente, el método **error( )** coloca una marca en la descarga para indicar que ocurrió un error. Los métodos **getUrl( )**, **getSize( )**, **getProgress( )** y **getStatus( )**regresan el valor de la variable correspondiente.

La clase ProgressRenderer

La clase **ProgressRender** es una pequeña clase de utilería que se utiliza para dibujar el avance de una descarga listada en la interfaz gráfica de la aplicación dentro de un componente **JTable**. Normalmente un objeto **JTable** dibuja el texto de cada celda como texto. Sin embargo, en ocasiones es útil dibujar el contenido de una celda de manera distinta. En la aplicación Download Manager, se va a dibujar el valor de avance de la descarga como una barra que represente el avance. La clase **ProgressRender** mostrada a continuación hace eso posible. Nótese que la clase extiende de **JProgressBar** e implementa **TableCellRender**:

```
import java.awt.*;
import javax.swing.*;
import javax.swing.table.*;
```

```
  // Esta clase dibuja un JProgressBar en la celda de una tabla
  class ProgressRenderer extends JProgressBar
    implements TableCellRenderer
  {
    // Constructor para ProgressRenderer.
    public ProgressRenderer(int min, int max) {
    super (min, max);
    }

  /* Devuelve el JProgressBar para ser utilizado como intérprete
     en la celda actual de la tabla */
  public Component getTableCellRendererComponent (
    JTable table, Object value, boolean isSelected,
    boolean hasFocus, int row, int column)
  {
    // Establece el porcentaje de avance en el JProgressBar
    setValue((int) ((Float) value).floatValue());
    return this;
  }
}
```

La clase **ProgressRender** toma ventaja del hecho que el componente **JTable** de Swing acepte extender su sistema de dibujo de celdas. Para extender el sistema de dibujo de celdas, primero, la clase **ProgressRender** implementa la interfaz **TableCellRender**. Segundo, una instancia de **ProgressRender** es registrada en una instancia de **JTable**; al hacer esto, se le indica a la instancia **JTable** cuales celdas deberán ser dibujadas con la extensión creada por el programador.

Implementar la interfaz **TableCellRender** requiere que la clase sobrescriba al método **getTableCellRenderComponent( )**. El método **getTableCellRenderComponent( )** se invoca cada vez que una instancia de la clase **JTable** dibuja una celda para la cual la clase con este método ha sido definida como responsable de hacer el trazo. A este método se le pasan diversas variables, pero en el caso anterior sólo se utiliza la variable **value**. La variable **value** contiene la información a colocar en la celda que está siendo dibujada, este valor es enviado al método **setValue( )** de **JProgressBar**. El método **getTableCellRenderComponent( )** concluye devolviendo una referencia a su clase. Esto funciona debido a que la clase **ProgressRender** es una subclase de **JProgressBar**, la cual desciende de la clase **Component** de AWT.

La clase DownloadsTableModel

La clase **DownloadsTableModel** aloja a la lista de descargas del **Download** Manager y es la fuente de datos para el componente **JTable** de la interfaz gráfica. La clase **DownloadTableModel** se muestra a continuación. Note que extiende a **AbstractTableModel** e implementa la interfaz **Observer**.

```
import java.util.*;
import javax.swing.*;
import javax.swing.table.*;

// Esta clase administra la tabla de datos de las descargas
class DownloadsTableModel extends AbstractTableModel
  implements Observer
{
```

```
// Estos son los nombres para las columnas de la tabla
private static final String[] columnNames = {"URL", "Tamaño",
  "Progreso", "Estado"};

// Estas son las clases de cada valor de columna
private static final Class[] columnClasses = {String.class,
  String.class, JProgressBar.class, String.class};

// Aquí se almacena la lista de descargas
private ArrayList<Download> downloadList new ArrayList<Download>();

// Añade una nueva descarga a la tabla
public void addDownload(Download download) {
  // Se registra para ser notificado cuando se presente algún cambio
  en la descarga.
  download.addObserver(this);

  downloadList.add(download);

  // Dispara una notificación de inserción de datos en la tabla
  fireTableRowslnserted(getRowCount() - 1, getRowCount() - 1);
}

// Devuelve un objeto tipo Download para el renglón especificado
public Download getDownload(int row) {
  return downloadList.get(row);
}

// Elimina una descarga de la lista
public void clearDownload(int row) {
  downloadList.remove(row);

  // Dispara una notificación de borrado de datos en la tabla
  fireTableRowsDeleted(row, row);
}

// Devuelve la cantidad de columnas en la tabla
public int getColumnCount() {
  return columnNames.length;
}

// Devuelve el nombre de una columna
public String getColumnName(int col) {
  return columnNames[col];
}

// Devuelve la clase de los datos en una columna
public Class getColumnClass(int col) {
  return columnClasses[col];
}

// Devuelve la cantidad de renglones en la tabla
public int getRowCount() {
  return downloadList.size();
}
```

```
  // Obtiene el valor de una celda
  public Object getValueAt(int row, int col) {
    Download download = downloadList.get(row);
    switch (col) {
      case 0: // URL
        return download.getUrl();
      case 1: // Size
        int size = download.getSize();
        return (size == -1) ? "" : Integer.toString(size);
      case 2: // Progress
        return new Float(download.getProgress());
      case 3: // Status
        return Download.STATUSES[download.getStatus()];
    }
    return "";
  }

  /* El método update es invocado cuando un objeto Download
     notifica a sus observadores de algún cambio */
  public void update(Observable o, Object arg) {
    int index = downloadList.indexOf(o);

    // Dispara una notificación de actualización de datos en la tabla
    fireTableRowsUpdated(index, index);
  }
}
```

La clase **DownloadTableModel** esencialmente es una clase de utilería utilizada por el objeto **JTable** para administrar los datos de la tabla. Cuando la instancia de **JTable** es inicializada se le relaciona con un objeto **DownloadTableModel**. Luego la instancia **JTable** llama a diversos métodos de **DownloadTableModel** para llenarse de información ella misma. El método **getColumnCount( )** es invocado para obtener el número de columnas en la tabla. De forma similar, el método **getRowCount( )** es utilizado para obtener el número de renglones en la tabla. El método **getColumnName( )** devuelve el nombre de una columna dado su identificador. El método **getDownload( )** toma un identificador de renglón y devuelve el objeto **Download** asociado. El resto de los métodos en la clase **DownloadTableModel** se detallan en las siguientes secciones.

El método addDownload( )

El método **addDownload**, mostrado abajo, añade un nuevo objeto **Download** a la lista de descargas y por ende un renglón en la tabla:

```
// Añade una nueva descarga a la tabla
public void addDownload(Download download) {
  // Se registra para ser notificado cuando se presente algún cambio
     en la descarga.
  download.addObserver(this);

  downloadList.add(download);

  // Dispara una notificación de inserción de datos en la tabla
  fireTableRowslnserted(getRowCount() - 1, getRowCount() - 1);
}
```

El método primero registra a la clase con el nuevo objeto **Download** como un **observador** interesado en recibir notificaciones de cambios. Luego, el objeto **Download** es añadido en la lista interna de descargas administradas. Finalmente, un evento de inserción de renglón en tabla se dispara para alertar a la tabla que un nuevo renglón ha sido agregado.

El método clearDownload( )

El método **clearDownload( )**, mostrado abajo, elimina un objeto **Download** de la lista de descargas administradas:

```
// Elimina una descarga de la lista
public void clearDownload(int row) {
downloadList.remove(row);

  // Dispara una notificación de borrado de datos en la tabla
  fireTableRowsDeleted(row, row);
}
```

Después de eliminar al objeto **Download** de la lista interna, se dispara un evento para notificar a la tabla que un renglón ha sido borrado.

El método getColumnClass ( )

El método **getColumnClass**, mostrado abajo, devuelve la clase de los datos en una columna específica.

```
// Devuelve la clase de los datos en una columna
public Class getColumnClass(int col) {
  return columnClasses[col];
}
```

Todas las columnas se despliegan como texto (esto es, como objetos de tipo **String**) excepto la columna"Progreso", la cual se despliega como una barra (esto es, como un objeto de tipo **JProgressBar**).

El método getValueAt( )

El método **getValueAt( )**, mostrado abajo, es llamado para obtener el valor que debe ser desplegado para cada celda en la tabla:

```
// Obtiene el valor de una celda
public Object getValueAt(int row, int col) {
  Download download = downloadList.get(row);
  switch (col) {
    case 0: // URL
      return download.getUrl();
    case 1: // Size
    int size = download.getSize();
      return (size == -1) ? "" : Integer.toString(size);
    case 2: // Progress
      return new Float(download.getProgress());
    case 3: // Status
```

```
      return Download.STATUSES[download.getStatus()];
  }
  return "";
}
```

Este método primero busca el objeto **Download** correspondiente al renglón especificado. Luego, la columna especificada se utiliza para determinar cual de los valores del objeto se debe devolver.

El método update( )

El método **update( )** mostrado abajo consuma la labor de la interfaz **Observer**. Este método le permite a la clase **DownloadTableModel** recibir notificaciones de los objetos **Download** cuando esos objetos cambian.

```
/* El método update es invocado cuando un objeto Download
   notifica a sus observadores de algún cambio */
public void update(Observable o, Object arg) {
  int index = downloadList.indexOf(o);

  // Dispara una notificación de actualización de datos en la tabla
  fireTableRowsUpdated(index, index);
}
```

A este método se le pasa una referencia al objeto **Download** que ha sufrido cambios. El objeto se pasa como un objeto de tipo **Observable**. Luego, un índice se busca el índice correspondiente a este objeto en la lista de descargas y con ese índice se genera un evento de notificación de actualización. El evento alerta a la tabla del cambio en uno de sus renglones. La tabla en respuesta redibujará el renglón cuyo índice es especificado en la variable index.

La clase DownloadManager

Ahora que las bases han sido colocadas con la explicación de cada una de las clases auxiliares de la aplicación Download Manager, podemos revisar a detalle la clase **DownloadManager**. La clase **DownloadManager** es responsable de crear y ejecutar la interfaz gráfica de la aplicación. Esta clase tiene un método **main( )**. El método **main( )** crea una instancia de la clase **DownloadManager** y luego invoca al método **show( )** de la instancia, lo cual causa que la interfaz gráfica aparezca en pantalla.

La clase **DownloadManager** se muestra abajo. Note que extiende de **JFrame** e implementa a la interfaz **Observer**. La siguiente sección examina la clase con detenimiento.

```
import java.awt.*;
import java.awt.event.*;
import java.net.*;
import java.util.*;
import javax.swing.*;
import javax.swing.event.*;

// La clase Download Manager
public class DownloadManager extends JFrame
  implements Observer
{
```

```
// Añade un campo de texto
private JTextField addTextField;

// Objeto que gestiona los datos de la tabla
private DownloadsTableModel tableModel;

// Tabla
private JTable table;

// Botones para indicar acciones sobre las descargas
private JButton pauseButton, resumeButton;
private JButton cancelButton, clearButton;

// Renglón actual
private Download selectedDownload;

// Bandera para indicar que el renglón seleccionado esta siendo eliminado
private boolean clearing;

// Constructor de la clase
public DownloadManager()
{
  // Asigna un título a la aplicación
  setTitle("Download Manager");

  // Establece el tamaño de la ventana
  setSize(640, 480);

  // Maneja el evento de cierre de la ventana
  addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent e) {
      actionExit();
    }
  }) ;

  // Colocar el menú de archivo
  JMenuBar menuBar = new JMenuBar();
  JMenu fileMenu = new JMenu("Archivo");
  fileMenu.setMnemonic(KeyEvent.VK_F);
  JMenultem fileExitMenultem = new JMenultem("Salir",
    KeyEvent. VK_X);
  fileExitMenultem.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      actionExit();
    }
  });
  fileMenu.add (fileExitMenultem);
  menuBar.add(fileMenu);
  setJMenuBar(menuBar);

  // Colocar un panel
  JPanel addPanel = new JPanel();
  addTextField = new JTextField(30);
```

```
addPanel.add(addTextField) ;
JButton addButton = new JButton("Añadir Descarga");
addButton.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    actionAdd();
  }
});
addPanel.add(addButton);

// Formar la tabla de descargas
tableModel = new DownloadsTableModel();
table = new JTable(tableModel);
table.getSelectionModel() .addListSelectionListener(new
  ListSelectionListener() {
  public void valueChanged(ListSelectionEvent e) {
    tableSelectionChanged();
  }
});
// Sólo permite seleccionar un renglón a la vez
table.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);

// Define a ProgressBar como el responsable de dibujar los elementos de la
columna Progress
ProgressRenderer renderer = new ProgressRenderer(0, 100);
renderer.setStringPainted(true); // muestra el avance
table.setDefaultRenderer(JProgressBar.class, renderer);

// Establece el alto del renglón para que el componente JProgressBar se
ajuste
table.setRowHeight(
  (int) renderer.getPreferredSize().getHeight());

// Colocar un panel
JPanel downloadsPanel = new JPanel();
downloadsPanel.setBorder(
  BorderFactory.createTitledBorder("Descargas"));
downloadsPanel.setLayout(new BorderLayout());
downloadspanel.add(new JScrollPane(table),
  BorderLayout.CENTER);

// Colocar el panel con los botones
JPanel buttonsPanel = new JPanel();
pauseButton = new JButton("Pausa");
pauseButton.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    actionpause();
  }
});
pauseButton.setEnabled(false);
buttonsPanel.add(pauseButton);
resumeButton = new JButton("Resume");
resumeButton.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    actionResume();
  }
});
```

```
    resumeButton.setEnabled(false);
    buttonsPanel.add(resumeButton);
    cancelButton = new JButton("Cancelar");
    cancelButton.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        actionCancel();
      }
    });
    cancelButton.setEnabled(false);
    buttonsPanel.add(cancelButton);
    clearButton = new JButton ("Limpiar");
    clearButton.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        actionClear();
      }
    });
    clearButton.setEnabled(false);
    buttonsPanel.add(clearButton);

    // Añadir los panel a pantalla
    getContentPane().setLayout(new BorderLayout());
    getContentPane().add(addPanel, BorderLayout.NORTH);
    getContentPane().add(downloadsPanel, BorderLayout.CENTER);
    getContentPane().add (buttonsPanel, BorderLayout.SOUTH);
  }

  // Salir del programa
  private void actionExit()
    System.exit(0);
  }

  // Añadir una nueva descarga
  private void actionAdd() {
    URL verifiedUrl = verifyUrl(addTextField.getText());
    if (verifiedUrl != null) {
      tableModel.addDownload(new Download(verifiedUrl));
      addTextField.setText(""); // limpia el campo de texto
    }else {
      JOptionPane.showMessageDialog(this,
        "URL no válido", "Error",
        JOptionPane.ERROR_MESSAGE);
    }
  }

  // Verificar dirección URL del archivo a descargar
  private URL verifyUrl(String url) {
    // Sólo permite URL con HTTP
    if (! url.toLowerCase().startsWith ("http://") )
      return null;

    // Verifica el formato del URL
    URL verifiedUrl = null;
    try {
```

```
    verifiedUrl = new URL(url);
    }catch (Exception e) {
      return null;
    }

    // Revisa que el URL haga referencia a un archivo
    if (verifiedUrl.getFile().length() < 2)
      return null;

    return verifiedUrl;
  }

  // Este método es invocado cuando cambia el renglón seleccionado
  private void tableSelectionChanged() {
    /* Expresar que ya no nos interesan las notificaciones de eventos
       del renglón anterior */
    if (selectedDownload != null)
      selectedDownload.deleteObserver(DownloadManager.this);

    /* Si no se está eliminando un renglón
       establecer un nuevo renglón como seleccionado y expresar
       interés por sus eventos. */
    if (!clearing && table.getSelectedRow() > -1)
      selectedDownload =
        tableModel.getDownload(table.getSelectedRow());
      selectedDownload.addObserver(DownloadManager.this);
      updateButtons();
    }
  }

  // Hacer una pausa en la descarga asociada al renglón seleccionado
  private void actionPause() {
    selectedDownload.pause();
    updateButtons();
  }

  // Continuar con la descarga asociada al renglón seleccionado
  private void actionResume() {
    selectedDownload.resume();
    updateButtons();
  }

  // Cancelar la descarga asociada al renglón seleccionado
  private void actionCancel() {
    selectedDownload.cancel();
    updateButtons();
  }

  // Borrar la descarga asociada al renglón seleccionado
  private void actionClear() {
    clearing = true;
    tableModel.clearDownload(table.getSelectedRow());
    clearing = false;
    selectedDownload = null;
    updateButtons();
  }
```

```
  /* Actualiza el estado de cada botón acorde con el
     estado de la descarga seleccionada */
  private void updateButtons() {
    if (selectedDownload != null) {
      int status = selectedDownload.getStatus();
      switch (status) {
        case Download.DOWNLOADING:
          pauseButton.setEnabled(true);
          resumeButton.setEnabled(false);
          cancelButton.setEnabled(true);
          clearButton.setEnabled(false);
          break;
        case Download.PAUSED:
          pauseButton.setEnabled(false);
          resumeButton.setEnabled(true);
          cancelButton.setEnabled(true);
          clearButton.setEnabled(false);
          break;
        case Download.ERROR:
          pauseButton.setEnabled(false);
          resumeButton.setEnabled(true);
          cancelButton.setEnabled(false);
          clearButton.setEnabled(true);
          break;
        default: // COMPLETE o CANCELLED
          pauseButton.setEnabled(false);
          resumeButton.setEnabled(false);
          cancelButton.setEnabled(false);
          clearButton.setEnabled(true);
      }
    } else {
      // Ningún renglón está seleccionado en la tabla
      pauseButton.setEnabled(false);
      resumeButton.setEnabled(false);
      cancelButton.setEnabled(false);
      clearButton.setEnabled(false);
    }
  }

  /* El método update se ejecuta cuando un objeto Download notifica
     a sus observadores de algún cambio. */
  public void update(Observable o, Object arg) {
    // actualizar los botones acorde a los cambios
    if (selectedDownload != null && selectedDownload.equals(o))
      updateButtons();
  }

// Ejecutar la aplicación
  public static void main(String[] args) {
```

```
    SwingUtilities.invokerLater(new Runnable() {
      public void run() {
        DownloadManager manager = new DownloadManager();
        manager.setVisible(true);
      }
    });
  }
}
```

Las variables de DownloadManager

La clase **DownloadManager** comienza con la declaración de diversas variables de instancia, muchas de las cuales almacenan referencias a controles de la interfaz gráfica. La variable **selectedDownload** almacena una referencia al objeto **Download** que representa el renglón seleccionado en la tabla. Finalmente, la variable de instancia **clearing** es de tipo **boolean** y sirve para indicar cuando una descarga está siendo eliminada de la tabla de descargas.

El constructor DownloadManager

Cuando se crea un objeto de la clase **DownloadManager**, todos los controles de la interfaz gráfica son inicializados por el constructor. El constructor contiene una gran cantidad de código, pero mucho de ese código es simple. A continuación una descripción rápida.

Primero, el título de la ventana se coloca realizando una llamada al método **setTitle( )**. Luego una llamada al método **setSize( )** establece el ancho y alto de la ventana en pixeles. Después de ello, se agrega un listener de eventos de ventana con el método **addWindowListener( )**, pasándole un objeto **WindowAdapter** que sobrescribe al método gestor de eventos **windowClosing( )**. Este método gestor llama al método **actionExit( )** cuando la ventana de la aplicación se cierra. Luego, una barra de menú con un menú "Archivo" se agrega a la ventana de la aplicación. En seguida se agrega un panel que contiene un campo de texto y un botón para que el usuario agregue descargas. Se agrega un **ActionListener** al botón para que el método **actionAdd( )** sea invocado cada vez que el botón sea presionado.

Después de lo anterior se construye la tabla de descargas. Un **ListSelectionListener** se agrega a la tabla para que cada vez que un renglón sea seleccionado en la tabla, el método **tableSelectionChanged( )** sea invocado. La forma en que los renglones pueden ser seleccionados en la tabla se actualiza a **ListSelectionModel.SINGLE_SELECTION** de manera que sólo un renglón pueda estar seleccionado a la vez. Limitar la selección de renglones a sólo uno simplifica la decisión de cuales botones deberán estar activos en la interfaz gráfica al momento de seleccionar determinado renglón. Después se crea una instancia de la clase **ProgressRender** y se relaciona con la tabla para que esta instancia se encargue de gestionar la columna "Progreso". La altura de los renglones en la tabla se modifica, llamando al método **table.setRowHeight( )**, para que sea la adecuada para visualizar el **ProgressRender**. Una vez que la tabla está lista se coloca dentro de un **JScrollPane** para agregarle barras de desplazamiento y luego es añadida a un panel.

Finalmente, se crea un panel con los botones de acción para la aplicación. Los botones incluyen las acciones de pausa, reiniciar, cancelar y eliminar. A cada uno de los botones se

le añade un **ActionListener** que invocará al método de acción respectivo cuando el botón sea presionado. Después de la creación del panel de botones, todos los paneles creados son colocados en la ventana.

El método verifyUrl( )

El método **verifyURL( )** es invocado por el método **actionAdd( )** cada vez que se agrega una nueva descarga en la aplicación. El método **verifyUrl( )** se muestra aquí:

```
// Verificar dirección URL del archivo a descargar
private URL verifyUrl(String url) {
  // Sólo permite URL con HTTP
  if (!url.toLowerCase().startsWith ("http://") )
    return null;

  // Verifica el formato del URL
  URL verifiedUrl = null;
  try {
    verifiedUrl = new URL(url);
  } catch (Exception e) {
    return null;
  }

  // Revisa que el URL haga referencia a un archivo
  if (verifiedUrl.getFile().length() < 2)
    return null;

  return verifiedUrl;
}
```

Este método primero verifica que el URL escrito por el usuario es un URL con el protocolo HTTP. En seguida el URL verificado se utiliza para construir un objeto de la clase **URL**. Si la dirección URL es incorrecta, el constructor de la clase **URL** genera una excepción. Finalmente, este método verifica que el URL corresponde a un archivo.

El método tableSelectionChanged( )

El método **tableSelectionChanged( )**, mostrado abajo, es invocado cada vez que un renglón es seleccionado en la tabla:

```
// Este método es invocado cuando cambia el renglón seleccionado
private void tableSelectionChanged() {
  /* Expresar que ya no nos interesan las notificaciones de eventos
     del renglón anterior */
  if (selectedDownload != null)
    selectedDownload.deleteObserver(DownloadManager.this);

  /* Si no se está eliminando un renglón
     establecer un nuevo renglón como seleccionado
     y expresar interés por sus eventos. */
  if (!clearing && table.getSelectedRow() > -1)
    selectedDownload =
      tableModel.getDownload(table.getSelectedRow());
```

```
    selectedDownload.addObserver(DownloadManager.this);
    updateButtons();
  }
}
```

Este método comienza revisando si ya existe un renglón seleccionado, para ello revisa si el valor de la variable **selectedDownload** es **null**. Si el valor de la variable **selectedDownload** no es **null** (existía un renglón seleccionado) entonces se declara que ya no son de interés para la clase los cambios en el **objetoDownload** que estuviera seleccionado. Luego se revisa la variable clearing. Si la tabla no está vacía y la variable **clearing** tiene un valor de **false**, entonces se actualiza el valor de la variable **selectedDownload** con una referencia al objeto **Download** equivalente al renglón seleccionado y el **DownloadManager** se registra como un observador del objeto **Download**.

Finalmente, se llama al método **updateButtons( )** para actualizar el estado de los botones de acuerdo al estado de la descarga seleccionada.

El método updateButtons()

El método **updateButtons( )** actualiza el estado de todos los botones en el panel de botones de acuerdo al estado de la descarga seleccionada en la tabla. El método **updateButtons( )** se muestra a continuación:

```
/* Actualiza el estado de cada botón acorde con el
   estado de la descarga seleccionada. */
private void updateButtons() {
  if (selectedDownload != null) {
    int status = selectedDownload.getStatus();
    switch (status) {
      case Download.DOWNLOADING:
        pauseButton.setEnabled(true);
        resumeButton.setEnabled(false);
        cancelButton.setEnabled(true);
        clearButton.setEnabled(false);
        break;
      case Download.PAUSED:
        pauseButton.setEnabled(false);
        resumeButton.setEnabled(true);
        cancelButton.setEnabled(true);
        clearButton.setEnabled(false);
        break;
      case Download.ERROR:
        pauseButton.setEnabled(false);
        resumeButton.setEnabled(true);
        cancelButton.setEnabled(false);
        clearButton.setEnabled(true);
        break;
      default: // COMPLETE o CANCELLED
        pauseButton.setEnabled(false);
        resumeButton.setEnabled(false);
```

```
            cancelButton.setEnabled(false);
            clearButton.setEnabled(true);
        }
      } else {
        // Ningún renglón está seleccionado en la tabla
        pauseButton.setEnabled(false);
        resumeButton.setEnabled(false);
        cancelButton.setEnabled(false);
        clearButton.setEnabled(false);
      }
    }
```

Si ninguna descarga está seleccionada en la tabla, todos los botones se deshabilitan, dejándolos con una apariencia grisácea. Sin embargo, si existe alguna descarga seleccionada, el estado de cada botón se establece con base al estado del objeto **Download** asociado a la descarga: **DOWNLOADING**, **PAUSED**, **ERROR**, **COMPLETE** o **CANCELLED**.

Gestión de los eventos de acción

Cada uno de los controles de la interfaz gráfica de la clase **DownloadManager** registra un **ActionListener** que invoca al respectivo método de acción. Los **ActionListener** se ejecutan cada vez que ocurre un evento que afecte al control. Por ejemplo, cuando un botón es presionado, se genera un **ActionEvent**. La forma en que trabajan los **ActionListener** es similar a la forma en que trabaja el patrón Observer descrito anteriormente.

Compilar y ejecutar el administrador de descarga

La aplicación **Download** Manager se compila como sigue:

```
javac DownloadManager.java DownloadsTableModel.java ProgressRender.java Down-
load.java
```

Y se ejecuta como sigue:

```
javaw DownloadManager
```

La aplicación es fácil de utilizar. Primero, escribimos el URL del archivo que deseamos descargar en el campo de texto que se encuentra en la parte superior de la pantalla. Por ejemplo, para descargar un archivo llamado 0072229713_code.zip del sitio Web de McGraw-Hill escribimos:

http://books.mcgraw-hill.com/downloads/products/0072229713/0072229713_code.zip

Este archivo contiene el código del libro "*The Art of Java*", escrito por Herbert Schildt y James Holmes.

Después de añadir una descarga al Download Manager, el usuario puede administrarla, seleccionándola en la tabla. Una vez seleccionada, es posible colocarla en pausa, cancelarla, reiniciarla o eliminarla. La Figura 33-2 muestra nuestra aplicación Download Manager en acción.

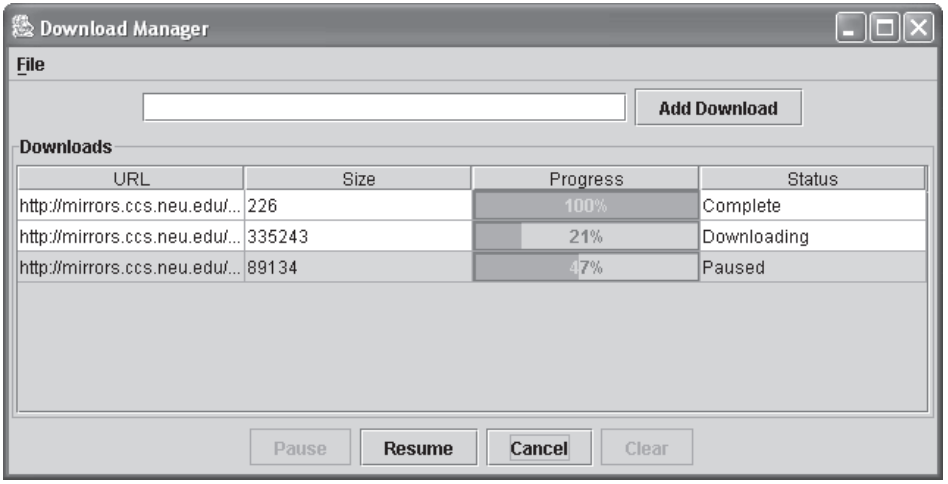


FIGURA 33-2 La aplicación Download Manager en acción

Mejorando el administrador de descargas

La aplicación descrita tal como está es completamente funcional, cuenta con la habilidad de detener y reiniciar descargas así como de descargar múltiples archivos al mismo tiempo; sin embargo, existen diversas mejoras que el lector puede intentar realizar por sí mismo. Veamos algunas ideas: soporte para servidor proxy, soporte para los protocolos FTP y HTTPS, soporte para "drag and drop". Una mejora especialmente llamativa es una agenda que nos permita iniciar una descarga a una hora específica, quizá a media noche cuando los recursos de la computadora no están siendo requeridos.

Nótese que las técnicas ilustradas en este capítulo no se limitan a la descarga de archivos en el sentido clásico. Existen muchos otros usos prácticos para el código descrito. Por ejemplo, muchos programas distribuidos por Internet vienen en dos partes. La primera parte es una aplicación compacta y pequeña que puede ser descargada rápidamente. Esta pequeña aplicación contiene un pequeño administrador de descargas responsable de descargar la segunda parte, la cual es generalmente mucho más grande. Este concepto es muy útil, especialmente con aplicaciones de gran tamaño, debido a que con ellas se incrementa la posibilidad de sufrir interrupciones. Adaptar nuestra aplicación de administración de descargas para el objetivo descrito antes puede ser también un excelente ejercicio.

APÉNDICE A

Usando los comentarios de documentación de Java

Como se explicó en la Parte I, Java maneja tres tipos de comentarios. Los primeros dos son los comentarios con // y los comentarios con /* */. El tercer tipo de comentario es conocido como *comentario de documentación*. Éste inicia con la secuencia de caracteres /** y termina con */. Los comentarios de documentación permiten incluir información sobre el programa dentro del propio programa. Luego, se puede utilizar la herramienta **javadoc** (proporcionada por el JDK) para extraer la documentación del programa y colocarla en un archivo HTML. Los comentarios de documentación hacen que la documentación de los programas sea más cómoda. Con seguridad usted ha visto documentación generada utilizando la herramienta **javadoc**, puesto que es la forma en que Sun documentó la API de Java.

Las etiquetas de javadoc

La aplicación **javadoc** reconoce las siguientes etiquetas:

Etiqueta	Significado
@author	Identifica al autor de la clase.
{@code}	Muestra información tal y como está, es decir sin procesarla con los estilos de HTML.
@deprecated	Especifica que una clase o un miembro de la clase está depreciado.
{@docRoot}	Especifica la ruta al directorio raíz de la documentación actual.
@exception	Identifica una excepción arrojada por un método.
{@inheritDoc}	Hereda un comentario de la superclase inmediata superior.
{@link}	Inserta un hipervínculo hacia otro tema.
{@linkplain}	Inserta un hipervínculo hacia otro tema, pero el hipervínculo es mostrado con una tipografía de texto plano.
{@literal}	Muestra información tal y como está, es decir sin procesarla en HTML.
@param	Documenta un parámetro de un método.
@return	Documenta el valor de retorno de un método.
@see	Especifica un hipervínculo a otro tema.

Etiqueta	Significado
@serial	Documenta un campo serializable por omisión.
@serialData	Documenta los datos escritos por los métodos **writeObject()** o **writeExternal()**.
@serialField	Documenta un componente de **ObjectStreamField**.
@since	Establece la versión en la cual se introdujo un cambio específico.
@throws	Realiza lo mismo que la etiqueta **@exception.**
{@value}	Muestra el valor de una constante, la cual tiene que ser un campo de tipo **static**.
@version	Especifica la versión de una clase.

Las etiquetas de documentación que comienzan con el signo arroba (@) son llamadas etiquetas *autónomas*, y deben utilizarse en su propia línea. Las etiquetas que comienzan con una llave ({), como **{@code}**, son llamadas etiquetas *en línea* y pueden ser utilizadas en descripciones más grandes. Se pueden utilizar también etiquetas HTML estándar en los comentarios de documentación. Sin embargo, algunas etiquetas HTML, como son las de encabezado, no deben ser utilizadas puesto que rompen con la estructura del archivo HTML generado por **javadoc**.

Se pueden utilizar comentarios de documentación para documentar clases, interfaces, campos, constructores y métodos. En todos los casos, los comentarios de documentación deberán preceder al elemento que está siendo documentado. Cuando se está documentando una variable, las etiquetas de documentación que se pueden utilizar son **@see**, **@serial**, **@serialField**, **{@value}**, y **@deprecated**. Para las clases y las interfaces, se pueden utilizar **@see**, **@author**, **@deprecated**, **@param**, y **@version**. Los métodos puede ser documentados con **@see**, **@return**, **@param**, **@deprecated**, **@throws**, **@serialData**, **{@inheritDoc}**, y **@exception**. Una etiqueta **{@link}**, **{@docRoot}**, **{@code}**, {**@literal**}, **@since**, o **{@linkplain}** puede ser utilizada en cualquier parte. A continuación se revisa cada una de las etiquetas.

@author

La etiqueta **@author** documenta al autor de una clase o interfaz. Esta etiqueta tiene la siguiente sintaxis:

@author *descripción*

Donde, *descripción* será normalmente el nombre del autor. Será necesario ejecutar **javadoc** con la opción **-author** para que el campo **@author** sea incluido en la documentación en HTML.

{@code}

La etiqueta **{@code}** permite incluir texto, como un pedazo de código, en un comentario. Este texto se muestra tal y cual con la tipografía del código, sin ningún procesamiento adicional, como la aplicación de estilo HTML. Esta etiqueta tiene la siguiente sintaxis:

{@code *pedazo-codigo*}

@deprecated

La etiqueta **@deprecated** especifica que una clase, interfaz o un miembro están en desuso. Se recomienda que se incluya la etiqueta **@see** o la etiqueta **{@link}** para informar al programador sobre las alternativas disponibles. La sintaxis es la siguiente:

@deprecated *descripción*

Donde, *descripción* es el mensaje que describe la depreciación. La etiqueta **@deprecated** puede ser utilizada en la documentación de variables, métodos, clases e interfaces.

{@docRoot}

{@docRoot} especifica la ruta del directorio raíz de la documentación actual.

@exception

La etiqueta **@exception** describe una excepción de un método. Tiene la siguiente sintaxis:

@exception *nombre-excepcion explicación*

Aquí, el nombre completo de la excepción es especificado por *nombre-excepcion* y explicación es una cadena que describe cómo puede ocurrir la excepción. La etiqueta **@exception** sólo puede ser utilizada en la documentación de un método.

{@inheritDoc}

Esta etiqueta hereda un comentario de la superclase inmediata superior.

{@link}

La etiqueta **{@link}** provee de un hipervínculo a información adicional. Esta etiqueta tiene la siguiente sintaxis:

{@link *pkg.class#miembro texto*}

Donde, *pkg.class#miembro* especifica el nombre de una clase o método al cual se le está añadiendo un hipervínculo, y texto es la cadena que será desplegada.

{@linkplain}

Inserta un hipervínculo en línea hacia otro tema. El hipervínculo es mostrado como texto plano. Por lo demás, es similar a **{@link}**.

{@literal}

La etiqueta **{@literal}** permite introducir texto en un comentario. Este texto es mostrado sin ningún formato ni ningún procesamiento en HTML. Esta etiqueta tiene la siguiente sintaxis.

{@literal *descripción*}

Donde, *descripción* es el texto ha ser incluido.

@param

La etiqueta **@param** documenta un parámetro de un método o el parámetro de tipo de una clase o interfaz. Esta etiqueta tiene la siguiente sintaxis:

@param *nombre-parámetero explicación*

Donde *nombre-parámetro* especifica el nombre de un parámetro. El significado de este parámetro es descrito por explicación. La etiqueta **@param** sólo puede ser utilizada en la documentación para un método, constructor, clase genérica o interfaz genérica.

@return

La etiqueta **@return** describe el valor que regresa un método. Esta etiqueta tiene la siguiente sintaxis:

@ return *explicación*

Donde, *explicación* describe el tipo y significado del valor que regresa el método. La etiqueta **@return** sólo puede ser utilizada en la documentación de un método.

@see

La etiqueta **@see** provee una referencia a información adicional. Sus usos más comunes se muestran aquí:

@see *ancla*
@see *pkg.clase#miembro texto*

En la primera forma, *ancla* es un hipervínculo a un URL ya sea absoluto o relativo. En la segunda forma *pkg.clase#miembro* especifica el nombre de un elemento, y *texto* es el texto mostrado para ese elemento. El parámetro texto es opcional, si no se utiliza, entonces el elemento especificado por *pkg.class#miembro* es mostrado. El nombre del miembro, también es opcional. Por lo tanto, se puede especificar una referencia a un paquete, clase o interfaz además de la referencia a un método o campo. El nombre puede estar especificado total o parcialmente. Sin embargo, el punto que precede al nombre del miembro (si éste existe) deberá ser remplazado por un carácter #.

@serial

La etiqueta **@serial** define el comentario para un campo serializable por omisión. Ésta tiene la siguiente sintaxis:

@serial *descripción*

Donde, *descripción* es el comentario para dicho campo.

@serialData

La etiqueta **@serialdata** documenta los datos escritos por los métodos **writeObject( )** y **writeExternal( )**. Tiene la siguiente sintaxis:

@serialData *descripción*

Donde, *descripción* es el comentario para dicho datos.

@serialField

Para una clase que implementa a la interfaz **Serializable**, la etiqueta **@serialField** provee comentarios para un componente **ObjectStreamField**. Esta etiqueta tiene la siguiente sintaxis:

@serialField *nombre tipo descripción*

Donde, *nombre* es el nombre del campo, *tipo* es su tipo y *descripción* es el comentario para dicho campo.

@since

La etiqueta **@since** establece que una clase o miembro fue incluido en una versión específica. Esta etiqueta tiene la siguiente sintaxis:

@since *versión*

Donde, *versión* es una cadena que define la versión en la cual el miembro o la clase fueron incluidos.

@throws

La etiqueta **@throws** tiene el mismo significado que la etiqueta **@exception**.

{@value}

La etiqueta **{@value}** tiene dos formas. La primera muestra el valor de la constante a la que precede, la cual debe de ser un campo **estático**. Esta etiqueta tiene la siguiente forma:

{@value}

La segunda variante muestra el valor de un campo **estático** específico. Esta forma se ve así:

{@value *pkg.clase#campo*}

Donde, *pkg.clase#campo* especifica el nombre del campo estático.

@version

La etiqueta **@version** especifica la versión de una clase. Esta etiqueta tiene la siguiente sintaxis:

@version *info*

Donde, *info* es una cadena que contiene la información de la versión, normalmente un número de versión, como 2.2. Es necesario especificar la opción **–version** cuando se ejecute la herramienta **javadoc**, para que el campo **@version** sea incluido en la documentación HTML.

Forma general de un comentario de documentación

Después del /** inicial, la primera línea o líneas son la descripción general de la clase, las variables o métodos. Después de esto, se incluyen una o más etiquetas @. Cada etiqueta de tipo @ debe estar al inicio de una nueva línea o estar después de uno o varios asteriscos (*) que están al inicio de una línea. Las etiquetas de un mismo tipo deberán ser agrupadas juntas. Por ejemplo, si se tienen tres etiquetas **@see**, éstas tienen que estar una después de otra. Las etiquetas en línea (aquellas que inician con una llave) pueden utilizarse en cualquier descripción.

Aquí se muestra un ejemplo de un comentario de documentación para una clase:

```
/**
* Esta clase dibuja una gráfica de barras
* @author Herbert Schildt
* @version 3.2
*/
```

Salida de javadoc

El programa **javadoc** toma como entrada un archivo fuente de un programa de Java y arroja varios archivos HTML que contienen la documentación del programa. La información sobre cada clase estará en su propio archivo HTML. **javadoc** también genera un índice y un árbol jerárquico. También se generaran otros archivos HTML.

Un ejemplo que utiliza comentarios de documentación

El siguiente es un ejemplo de un programa que utiliza comentarios de documentación. Note la forma en como cada comentario precede al elemento que describe. Después de ser procesado por **javadoc**, la documentación de la clase **SquareNum** estará disponible en el archivo **SquareNum.html**.

```
import java.io.*;

/**
* Esta clase muestra el uso de comentarios de documentación.
* @author Herbert Schildt
* @version 1.2
*/
public class SquareNum
    /**
    * Este método regresa el cuadrado de un número.
    * Esta es una descripción multilínea. Es posible usar
    * tantas líneas como se desee.
    * @param num El valor que será elevado al cuadrado.
    * @return num al cuadrado
  */
  public double square(double num) {
  return num * num;
  }

  /**
    * Este método lee un valor dado por usuario
    * @return El valor introducido como un valor de tipo double
    * @exception IOException se genera si existe un error en la entrada de datos
    * @see IOException
  */
  public double getNumber() throws IOException {
    // crea un BufferedReader usando System.in
    InputStreamReader isr = new InputStreamReader(System.in);
    BufferedReader inData = new BufferedReader(isr);
    String str;

    str = inData.readLine();
    return (new Double(str)).doubleValue();
  }
  /**
    * Este método invoca a square()
    * @param args No se utiliza.
    * @exception IOException Se genera si existe un error en la entrada de datos
    * @see IOException
  */

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

  {
    SquareNum ob = new SquareNum();
    double val;

    System.out.println("Escriba el valor del cual se calculará el cuadrado: ");
    val = ob.getNumber();
    val = ob.square(val);

    System.out.println("El cuadrado es: " + val);
  }
}
```

Índice

B

C

D

E

F

G

H

K

L

M

N

O

P

S

T

W

X

Y

Z