¡Desarrolle sus habilidades en programación!

Fundamentos de Java

Tercera edición

Actualizado para J2SE 5

CÓDIGO GRATUITO EN LÍNEA

www.mcgraw-hill-educacion.com

Herbert Schildt

Mc Graw Hill

Fundamentos de Java™

3ª. edición

Fundamentos de Java
3ª. edición

Herbert Schildt

Traducción
Eloy Pineda Rojas
Traductor profesional

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

Gerente de división: Fernando Castellanos Rodríguez
Editor de desarrollo: Cristina Tapia Montes de Oca
Supervisor de producción: Jacqueline Brieño Alvarez

Fundamentos de Java 3ª. edición.

McGraw-Hill Interamericana

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 : 970-10-5930-1

Translated from the third English edition of
JAVA A BEGINNER'S GUIDE
By: Herbert Schildt

ISBN: 0-07-223189-0

1234567890 0987543216

Impreso en México Printed in Mexico

Acerca del autor

Herbert Schildt es autor y líder mundial en la creación de libros de programación. Es también autoridad en los lenguajes C, C++, Java y C#, y maestro en programación en Windows. Sus libros de programación han vendido más de tres millones de copias en todo el mundo y se han traducido a todos los lenguajes importantes. Es autor de numerosos bestsellers, dentro de los que se incluyen, *Java: The Complete Reference, C: The Complete Reference* y *C#: The Complete Reference*. Además, es coautor de The Art of Java. Schildt cuenta con un título universitario y posgrado por parte de la Universidad de Illinois. Puede contactar al autor en su oficina al (217) 586-4683 en Estados Unidos. Su sitio Web es **www.HerbSchildt.com**.

Resumen del contenido

Contenido

Prefacio

Java es el lenguaje más importante de Internet. Más aún, es el lenguaje universal de los programadores Web en todo el mundo. Para ser un desarrollador Web profesional es necesario dominar Java. Por lo tanto, si su futuro se encuentra en la programación en Internet, ha elegido el lenguaje correcto; y este libro le ayudará a aprenderlo.

El objetivo es enseñarle los fundamentos de la programación en Java. En este libro encontrará un método que se desarrolla paso a paso, junto con numerosos ejemplos, pruebas de autoevaluación y proyectos. Debido a que se parte de la premisa de que usted no cuenta con experiencia previa en programación, el libro inicia con los fundamentos, es decir, con la manera, por ejemplo, de compilar y ejecutar un programa en Java. Luego se analiza cada palabra clave en este lenguaje y se concluye con algunas de las características más avanzadas de Java, como la programación con varios subprocesos, las opciones genéricas y los applets. Hacia el final del libro, usted tendrá un firme conocimiento de los elementos esenciales de la programación en Java.

Es importante establecer desde el principio que este libro sólo representa un punto de partida. Java va más allá de los elementos que definen el lenguaje pues incluye también amplias bibliotecas y herramientas que ayudan al desarrollo de programas. Más aún, Java proporciona un conjunto sofisticado de bibliotecas que manejan la interfaz de usuario del explorador. Para ser un programador profesional en Java se requiere también el dominio de estas áreas. Al finalizar este libro, usted tendrá los conocimientos para dar seguimiento a cualquier otro aspecto de Java.

La evolución de Java

Sólo unos cuantos lenguajes han cambiado de manera importante la esencia de la programación. En este selecto grupo, uno de ellos se destaca debido a que su impacto fue rápido y de gran alcance. Este lenguaje es, por supuesto, Java. No resulta exagerado afirmar que el lanzamiento original de Java 1.0 en 1995 por parte de Sun Microsystems causó una revolución en la programación que trasformó de manera radical Web y lo convirtió en un entorno enormemente interactivo. En el proceso, Java estableció un nuevo estándar en el diseño de lenguajes para computadoras.

Con los años, Java siguió creciendo, evolucionando y redefiniéndose en distintas formas. A diferencia de muchos otros lenguajes que se muestran lentos para incorporar nuevas características, Java ha estado de manera continua al frente del diseño de lenguaje para computadoras. Una razón de ello es la cultura de innovación y cambio que lo ha rodeado. Como resultado, este lenguaje ha recorrido varias actualizaciones (algunas relativamente pequeñas y otras de mayor importancia).

La primera actualización importante de Java fue la versión 1.1. Las funciones agregadas en Java 1.1 fueron más sustanciales de lo que se pensaría a partir del pequeño aumento en el número de versión. Por ejemplo, Java 1.1 agregó muchos elementos nuevos de biblioteca, redefinió la manera en que se manejaban los eventos y reconfiguró muchas características de la biblioteca original de la versión 1.0.

La siguiente versión importante de Java fue Java 2, donde el 2 indicaba "segunda generación". La creación de Java 2 constituyó un parteaguas y marcó el inicio de la "era moderna" de Java. La primera versión de Java 2 llevó el número de versión 1.2. Resulta extraño que la primera versión de Java 2 utilizara el número de versión 1.2. El número aludía originalmente a la versión interna de las bibliotecas de Java, pero luego se generalizó para aludir a toda la versión. Con Java 2, Sun reempaquetó el producto Java como J2SE (Java 2 Platform Standard Edition) y el número de versión empezó a aplicarse a ese producto.

La siguiente actualización de Java fue J2SE 1.3. Esta versión de Java fue la primera actualización importante de la versión original de Java 2, pues, en su mayor parte, contenía adiciones a las funciones existentes y le "apretó las tuercas" al entorno de desarrollo. La versión de J2SE 1.4 mejoró Java aún más. Esta versión contenía nuevas e importantes funciones, incluidas las excepciones encadenadas, la E/S de canal y la palabra clave **assert**.

La última versión de Java es la J2SE 5. Aunque cada una de las actualizaciones anteriores de Java ha sido importante, ninguna se compara en escala, tamaño y alcance con la J2SE 5. ¡Ésta ha cambiado de manera fundamental el mundo de Java!

J2SE 5: la segunda revolución de Java

Java 2 Platform Standard Edition versión 5 (J2SE 5) marca el inicio de la segunda revolución de Java. J2SE 5 agrega muchas funciones nuevas a Java que cambian de manera fundamental el carácter del lenguaje aumentando su capacidad y su alcance. Estas adiciones son tan profundas que modificarán para siempre la manera en que se escribe el código de Java. No se puede pasar por alto la fuerza revolucionaria de J2SE 5. Para darle una idea del alcance de los cambios originados por J2SE 5, he aquí una lista de las nuevas características importantes que se cubren en este libro.

- Elementos genéricos
- Autoencuadre/desencuadre
- Enumeraciones
- El bucle mejorado **for** del estilo "for-each"
- Argumentos de longitud variable (varargs)
- Importación estática
- Metadatos (anotaciones)

No se trata de una lista de cambios menores o actualizaciones incrementales. Cada elemento de la lista representa una adición importante al lenguaje Java. Algunos, como los elementos genéricos, el **for** mejorado y los varargs introducen nuevos elementos de sintaxis. Otros, como el autoencuadre y el autodesencuadre, modifican la semántica del lenguaje. Los metadatos agregan una dimensión completamente nueva a la programación. En todos los casos, se han agregado funciones nuevas y sustanciales.

La importancia de estas nuevas funciones se refleja en el uso de la versión número 5. Normalmente, el número de versión siguiente para Java habría sido 1.5; sin embargo, los cambios y las nuevas funciones resultan tan importantes que un cambio de 1.4 a 1.5 no expresaría la magnitud de éste. En lugar de ello, Sun decidió llevar el número de versión a 5, como una manera de destacar que un evento importante se estaba generando. Así, el producto actual es el denominado J2SE 5 y el kit para el desarrollador es el JDK 5. Sin embargo, con el fin de mantener la consistencia, Sun decidió usar 1.5 como *número interno de la versión*. De ahí que 5 sea el número externo de la versión y 1.5 sea el interno.

Debido a que Sun usa el 1.5 como número interno de la versión, cuando le pida al compilador su versión, éste responderá con 1.5 en lugar de 5. Además, la documentación en línea proporcionada por Sun utiliza 1.5 para aludir a las funciones agregadas en el J2SE 5. En general, cada vez que vea 1.5, simplemente significa 5.

Se ha actualizado por completo este libro a fin de que incluya las nuevas funciones agregadas en J2SE 5. Para manejar todos los nuevos materiales, se agregaron dos módulos completamente nuevos a esta edición. En el módulo 12 se analizan las enumeraciones, el autoencuadre, la importación estática y los metadatos, mientras que en el módulo 13 se examinan los elementos genéricos. Las descripciones del bucle **for** del estilo "for-each" y los argumentos de longitud variable se integraron en los módulos existentes.

Cómo está organizado este libro

En este libro se presenta un tutorial en el que los avances se producen a un ritmo constante y en el que cada sección parte de lo aprendido en la anterior. Contiene 14 módulos, y en cada uno de ellos se analiza un aspecto de Java. Este libro es único porque incluye varios elementos especiales que refuerzan lo que usted ha aprendido.

Habilidades fundamentales

Cada módulo empieza con un conjunto de las habilidades fundamentales que usted aprenderá y se indica, además, la ubicación de cada habilidad.

Revisión de habilidades dominadas

Cada módulo concluye con una revisión del dominio de las habilidades, es decir, con una prueba de autoevaluación que le permite probar sus conocimientos. Las respuestas se presentan en el apéndice A.

Revisión de los avances

Al final de cada sección importante, se presenta una revisión de los avances mediante la cual probará su comprensión de los puntos clave de la sección anterior. Las respuestas a estas preguntas se encuentran en la parte inferior de la página.

Pregunte al experto

Dispersos por todo el libro se encuentran recuadros especiales de "Pregunte al experto". Éstos contienen información adicional o comentarios importantes acerca de un tema, y emplean un formato de preguntas y respuestas.

Proyectos

Cada módulo contiene uno o más proyectos que le muestran cómo aplicar lo aprendido. Se trata de ejemplos realistas que podrá usar como puntos de partida para sus propios programas.

No se necesita experiencia previa en programación

En este libro no se parte de la premisa de que usted cuenta con experiencia previa en programación. Por lo tanto, podrá consultar este libro aunque nunca antes haya programado. Sin embargo, si cuenta con cierta experiencia en programación, avanzará un poco más rápido. Asimismo, tenga en cuenta que Java es diferente, en varios sentidos, a otros lenguajes populares de programación, así que es importante que no vaya directamente a las conclusiones. Por consiguiente, se le aconseja, aun a los programadores experimentados, una lectura cuidadosa.

Software necesario

Para ejecutar y compilar los programas de este libro, necesitará la versión más reciente del kit de desarrollo de Java (Java Development Kit, JDK) de Sun que, al momento de la publicación del presente libro, era Java 2 Platform Standard Edition, versión 5 (J2SE 5). En el módulo 1 se proporcionan las instrucciones para obtenerlo.

Si está usando una versión anterior de Java, como J2SE 1.4, podrá usar aún este libro, pero no podrá compilar y ejecutar los programas que usan las nuevas funciones agregadas en J2SE 5.

No lo olvide: el código está en Web

Recuerde, el código fuente de todos los ejemplos y proyectos de este libro están disponibles, sin costo, en el sitio Web **www.mcgraw-hill-educación.com**.

Para conocer aún más

Fundamentos de Java es su puerta a la serie de libros de programación de Herb Schildt. He aquí algunos otros libros que podrían resultarle de interés.

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

- *Java: The Complete Reference, J2SE 5 Edition.*
- *The Art of Java*

Para aprender acerca de C++, estos libros le resultarán especialmente útiles:

- *C++: The Complete Reference*
- *Teach Yourself C++*
- *C++ from the Ground Up*
- *STL Programming from the Ground Up*
- *The Art of C++*

Para aprender acerca de C#, sugerimos:

- *C#: A Beginner's Guide*
- *C#: The Complete Reference*

Si quiere aprender más acerca del lenguaje C, entonces los siguientes títulos le interesarán:

- *C: The Complete Reference*
- *Teach Yourself C*

Cuando necesite de respuestas sólidas, consulte a Herbert Schildt, la autoridad más reconocida en programación.

Módulo 1

Fundamentos de Java

HABILIDADES FUNDAMENTALES

1.1	Conozca la historia y filosofía de Java
1.2	Comprenda la contribución de Java a Internet
1.3	Comprenda la importancia del código de bytes
1.4	Conozca la terminología de Java
1.5	Comprenda los principios fundamentales de la programación orientada a objetos
1.6	Cree, compile y ejecute un programa simple de Java
1.7	Use variables
1.8	Use las instrucciones de control **if** y **for**
1.9	Cree bloques de código
1.10	Comprenda la manera en la que se posicionan, sangran y terminan las instrucciones
1.11	Conozca las palabras clave de Java
1.12	Comprenda las reglas para los identificadores de Java

El crecimiento de Internet y World Wide Web cambiaron de manera fundamental la forma de la computación. En el pasado el ciber paisaje estaba dominado por las PC independientes y aisladas. Hoy día, casi todas las PC están conectadas con Internet que, por sí mismo, se ha transformado pues al principio ofrecía una manera conveniente de compartir archivos e información y en la actualidad se ha convertido en un universo computacional enorme y distribuido. Con estos cambios, una nueva manera de programar surgió: Java.

Java es más que un lenguaje importante de Internet: revolucionó la programación cambiando la manera en la que pensamos acerca de la forma y la función de un programa. Hoy día, para ser un programador profesional se requiere la capacidad de programar en Java: así de importante es. A medida que avance en la lectura de este libro, aprenderá las habilidades necesarias para dominarlo.

El objetivo de este módulo es proporcionar una introducción a Java, incluyendo su historia, su diseño, su filosofía y varias de sus características más importantes. Por mucho, lo más difícil acerca del aprendizaje de un lenguaje de programación es el hecho de que no existen elementos aislados, sino que los componentes del lenguaje funcionan de manera conjunta. Esta interrelación resulta especialmente significativa en Java. En realidad, es difícil analizar un aspecto de Java sin incluir otros aspectos. Para ayudar a superar este problema, en este módulo se proporciona una breve presentación general de las funciones de Java como, por ejemplo, la forma general de un programa, algunas estructuras básicas de control y los operadores. Aunque no se profundizará mucho, enfatizarán los conceptos generales que le son comunes a cualquier programa de Java.

HABILIDAD FUNDAMENTAL 1.1

Los orígenes de Java

La innovación en los lenguajes computacionales está determinada por dos factores: mejoras en el arte de la programación y cambios en el entorno del cómputo. Java no es la excepción. Aprovechando la rica herencia de C y C++, Java ofrece un mayor refinamiento y funciones que reflejan el estado actual del arte de la programación. Respondiendo al surgimiento del entorno en línea, Java ofrece funciones que modernizan la programación con el fin de desarrollar una arquitectura altamente distribuida.

James Gosling, Patrick Naughton, Chris Warth, Ed Frank y Mike Sheridan concibieron Java en Sun Microsystems en 1991. A este lenguaje se le llamó inicialmente “Oak” pero se le renombró “Java” en 1995. Sorpresivamente, ¡Internet no fue el objetivo original de Java! La motivación principal, en cambio, fue la necesidad de un lenguaje que fuera independiente de la plataforma y que pudiera emplearse para crear un software que estuviera incrustado en varios dispositivos electrónicos para uso del consumidor, como tostadoras, hornos de microondas y controles remotos. Como usted ya probablemente adivinó, se usan muchos tipos diferentes de CPU como controladores. El problema es que la mayor parte de los lenguajes de cómputo están diseñados para compilarse con un destino específico. Pensemos, por ejemplo, en C++.

Aunque es posible compilar una página de C++ para cualquier tipo de CPU, para ello se requiere un compilador completo de C++ orientado a ese CPU. Sin embargo, el problema es que los compiladores son caros y su creación requiere mucho tiempo. En el intento de encontrar una mejor solución, Gosling y sus demás compañeros trabajaron en un lenguaje portable, de plataforma cruzada, que pudiera producir un código que se ejecutara en diversos CPU bajo entornos diferentes. Este esfuerzo culminó en la creación de Java.

Por la época en la que se trabajaba en los detalles de Java, surgió un segundo factor que resultó más importante y que desempeñaría un papel crucial en el futuro de Java. Por supuesto, esta segunda fuerza fue World Wide Web. En el caso de que Web no hubiera tomado forma casi al mismo tiempo en que se dio la implementación de Java, éste último habría conservado su utilidad pero se habría mantenido como un lenguaje oscuro para la programación de los aparatos electrónico antes mencionados. Sin embargo, con el surgimiento de Web, Java fue impulsado al primer plano del diseño de lenguajes de cómputo, debido a que Web exigía también programas portables.

La mayoría de los programadores aprenden al principio de su carrera que los programas portables son tan evasivos como deseables. Mientras que el desafío de hallar una manera de crear programas eficientes y portables (independientes de la plataforma) es casi tan antiguo como la propia disciplina de la programación, dicho desafío había pasado a un segundo plano en relación con otros problemas más relevantes. Sin embargo, con el surgimiento de Internet y Web, el viejo problema de la portabilidad volvió para tomar revancha. Después de todo, Internet es un universo diverso, distribuido y poblado con muchos tipos de computadoras, sistemas operativos y CPU.

Hacia 1993 resultó obvio para los miembros del equipo de diseño de Java que los problemas de portabilidad que comúnmente surgen cuando se crea un código para controladores incrustados, surgen también al momento de tratar de crear un código para Internet. Este descubrimiento logró que el centro de atención de Java cambiara de los aparatos electrónicos para el consumidor a la programación para Internet. De este modo, aunque el deseo de desarrollar un lenguaje de programación de arquitectura neutral constituyó la chispa inicial, Internet fue el que finalmente condujo al éxito de Java a gran escala.

Cómo se relaciona Java con C y C++

Java está relacionado directamente con C y C++. Hereda su sintaxis de C y su modelo de objeto está adaptado a partir de C++. La relación de Java con C y C++ es importante por varias razones. En primer lugar, muchos programadores están familiarizados con la sintaxis de C, C++, o ambos. Este hecho le facilita a un programador de C o C++ aprender Java y, de igual manera, a un programador de Java aprender C o C++.

En segundo lugar, los diseñadores de Java no "reinventaron la rueda", sino que refinaron aún más un paradigma de programación que había tenido mucho éxito. La época moderna de la programación empezó con C. Cambió a C++ y ahora a Java. Al heredar y basarse en esa rica herencia, Java proporciona un entorno de programación poderoso y lógicamente consistente que toma lo mejor del pasado y agrega nuevas funciones que el entorno en línea requiere. Tal vez lo más importante sea que, debido a sus similitudes, C, C++ y Java definen un marco de trabajo conceptual común para el programador profesional. Los programadores no enfrentan mayores fisuras cuando cambian de un lenguaje a otro.

Una de las filosofías centrales de C y C++ en cuanto al diseño es que el programador es la persona que tiene el control. Java hereda también dicha filosofía. Con excepción de las restricciones impuestas por el entorno de Internet, Java le proporciona a usted, es decir, el programador, un control total. Si usted programa bien, sus programas así lo reflejarán. Si programa de manera deficiente, sus programas igualmente lo reflejarán. En otras palabras, Java no es un lenguaje para aprender: es un lenguaje para programadores profesionales.

Java cuenta con otro atributo en común con C y C++: fue diseñado, probado y afinado por programadores reales y en activo. Es un lenguaje surgido de las necesidades y la experiencia de la gente que lo concibió. En este sentido, no existe una mejor manera de producir un lenguaje profesional de altos vuelos.

Debido a las similitudes entre Java y C++, sobre todo en el soporte que brindan a la programación orientada a objetos, resulta tentador pensar en Java como la simple "versión de C++ para Internet". Sin embargo, ello sería un error pues Java tiene importantes diferencias prácticas y filosóficas. Aunque fue influido por C++, no es una versión mejorada de éste (por ejemplo, no es compatible ni hacia arriba ni hacia abajo con C++). Por supuesto, las similitudes con C++ son significativas; por lo tanto, si usted es un programador de C++, se sentirá como en casa con Java. Otro punto: Java no fue diseñado para reemplazar a C++: fue diseñado para resolver un cierto conjunto de problemas, mientras que C++ fue diseñado para resolver otro conjunto diferente. Ambos coexistirán durante muchos años más.

Cómo se relaciona Java con C#

Recientemente ha llegado a escena un nuevo lenguaje llamado C#. Creado por Microsoft para dar soporte a su .NET Framework, C# está íntimamente relacionado con Java. En realidad, muchas de las funciones de C# se adaptaron directamente de Java. Tanto Java como C# comparten una misma sintaxis general semejante a C++, soportan la programación distribuida y utilizan el mismo modelo de objeto. Por supuesto, hay diferencias entre Java y C#, pero el aspecto y el manejo de estos lenguajes es muy similar. Esto significa que si ya conoce C#, le será especialmente fácil aprender Java. De manera que si va a utilizar C# en el futuro, entonces su conocimiento de Java le será útil.

Dada la similitud entre Java y C#, parece natural preguntar, "¿C# reemplazará a Java?" La respuesta es No. Java y C# están optimizados para dos tipos diferentes de entornos de cómputo. Así como C++ y Java coexistirán por mucho tiempo, también lo harán C# y Java.

Comprobación de avance

1. Java es útil para Internet porque puede producir programas _____.

2. ¿De cuáles lenguajes desciende directamente Java?

1. portables
2. C y C++.

HABILIDAD FUNDAMENTAL 1.2

La contribución de Java a Internet

Internet ayudó a catapultar a Java al primer plano de la programación, y Java, a su vez, ha tenido un profundo efecto en Internet. La razón es muy simple: Java expande el universo de los objetos que pueden desplazarse libremente por el ciberespacio. En una red hay dos categorías muy amplias de objetos que se transmiten entre el servidor y su computadora personal: información pasiva y programas dinámicos y activos. Por ejemplo, cuando lee su correo electrónico está viendo datos pasivos. Aunque descargue un programa, el código de éste sólo contendrá datos pasivos hasta que lo ejecute. Sin embargo, es posible transmitir a su computadora un segundo tipo de objeto: un programa dinámico, que se autoejecute. Este tipo de programa constituye un agente activo en la computadora cliente, pero es iniciado por el servidor. Por ejemplo, el servidor podría proporcionar un programa para desplegar apropiadamente los datos que está enviando.

Aunque los programas en red son deseables y dinámicos, también presentan problemas serios en las áreas de seguridad y portabilidad. Antes de Java, el ciberespacio estaba totalmente cerrado para la mitad de las entidades que ahora viven allí. Como verá, Java atiende estas preocupaciones y, al hacerlo, ha definido una nueva forma de programa: el applet.

Los applets de Java

Un *applet* es un tipo especial de programa de Java que está diseñado para transmitirse en Internet y que se ejecuta automáticamente en un explorador Web compatible con Java. Más aún, un applet se descarga bajo demanda, como cualquier imagen, archivo de sonido o clip de video. La diferencia más importante es que un applet es un *programa inteligente*, no sólo una animación o un archivo multimedia. En otras palabras, un applet es un programa que puede reaccionar a las entradas del usuario y cambiar dinámicamente (no sólo ejecutar la animación y el sonido una y otra vez).

Si bien los applets de Java son excitantes, sólo serían ideas deseables si Java no atendiera dos problemas fundamentales asociados con ellos: la seguridad y la portabilidad. Antes de seguir adelante, definamos lo que estos dos términos significan en relación con Internet.

Seguridad

Como seguramente ya lo sabe, cada vez que descarga un programa "normal", se pone en riesgo de una infección por virus. Antes de Java, la mayoría de los usuarios no descargaban programas ejecutables con frecuencia y, quienes lo hacían, revisaban que éstos no tuvieran virus antes de su ejecución. Aún así, la mayoría de los usuarios se preocupaban todavía por la posibilidad de la infección de sus sistemas con un virus o por permitir que programas malintencionados se ejecutaran libremente en sus sistemas. (Un programa malintencionado puede recolectar información privada, como números de tarjetas de crédito, saldos de cuentas bancarias y contraseñas al revisar el contenido del sistema de archivos de su computadora.) Java responde a estas preocupaciones al proporcionar un *firewall* entre una aplicación en red y su computadora.

Cuando usa un explorador Web compatible con Java, es posible descargar applets de Java de manera segura, sin miedo a una infección por virus. La manera en la que Java lo logra es mediante la confinación de un programa de Java al entorno de ejecución de Java y el impedimento que impone de acceder a otras partes de la computadora. (En breve verá cómo se logra esto.) Francamente, la capacidad de descargar applets con la confianza de que no dañará la computadora cliente constituye el aspecto más significativo de Java.

Portabilidad

Como se analizó antes, muchos tipos de computadoras y sistemas operativos están conectados con Internet. Para que los programas se descarguen dinámicamente a todos los tipos distintos de plataformas, se necesitan algunos medios para generar un código ejecutable que sea portable. Como verá pronto, el mismo mecanismo que ayuda a establecer la seguridad también ayuda a crear la portabilidad. Por supuesto, la solución de Java a estos dos problemas resulta refinada y eficiente.

HABILIDAD FUNDAMENTAL 1.3

La magia de Java: el código de bytes

La clave que permite a Java resolver los problemas de seguridad y portabilidad que se acaban de describir es que la salida de un compilador de Java *no es* un código ejecutable, sino un código de bytes. El *código de bytes* es un conjunto altamente optimizado de instrucciones diseñado para que sea ejecutado por el sistema de Java en tiempo de ejecución. A dicho sistema se le denomina máquina virtual de Java (Java Virtual Machina, JVM), es decir, la máquina virtual de Java es un *intérprete de código de bytes*. Esto puede resultarle un poco sorprendente. Como sabe, la mayor parte de los lenguajes modernos, como C++, están diseñados para la compilación, no la interpretación (sobre todo debido a problemas de desempeño). Sin embargo, el hecho de que un programa de Java se ejecute en la JVM ayuda a resolver los principales problemas relacionados con los programas descargados en Internet. He aquí por qué.

La traducción de un programa de Java en código de bytes facilita la ejecución de un programa en una gran variedad de entornos. La razón es sencilla: sólo la máquina virtual de Java necesita implementarse en cada plataforma. Una vez que existe el paquete en tiempo de ejecución para un sistema dado, cualquier programa de Java puede ejecutarse en él. Recuerde que, a pesar de que los detalles de la JVM serán diferentes entre plataformas, todas las JVM comprenden el mismo código de bytes de Java. Si se compilara un programa de Java en código nativo, entonces tendrían que existir diferentes versiones del mismo programa para cada tipo de CPU conectado a Internet. Claro está que ésta no es una solución factible. Por lo tanto, la interpretación del código de bytes es la manera más fácil de crear programas realmente portables.

El hecho de que un programa de Java sea interpretado ayuda también a que sea seguro. Debido a que la ejecución de todos los programas de Java está bajo el control de la JVM, ésta puede contener al programa y evitar que genere efectos colaterales fuera del sistema. La seguridad se mejora también con ciertas restricciones del lenguaje.

Por lo general, cuando se interpreta un programa, éste se ejecuta de manera sustancialmente más lenta de lo que lo que se ejecutaría si se compilara en código ejecutable. Sin embargo, con Java la diferencia

entre ambos códigos no es muy grande: el uso de un código de bytes permite que el sistema de Java en tiempo de ejecución ejecute el programa mucho más rápido de lo que se esperaría.

Aunque Java se diseñó para la interpretación, técnicamente nada impide que compile al vuelo el código de bytes en código nativo. Por tal motivo, Sun empezó proporcionando su tecnología HotSpot poco después del lanzamiento inicial de Java. HotSpot proporciona un compilador JIT (Just In Time, justo a tiempo) para el código de bytes. Cuando un compilador JIT es parte de la JVM compila, en tiempo real, el código de bytes en código ejecutable, parte por parte, de acuerdo con la demanda. Es importante comprender que no es posible compilar todo un programa de Java en código ejecutable de una sola vez debido a que Java realiza varias comprobaciones que sólo pueden realizarse en tiempo de ejecución. En cambio, el JIT compila el código conforme se requiera durante la ejecución. Más aún, no todas las secuencias del código de bytes están compiladas (sólo aquéllas que se beneficiarán con la compilación). El código restante simplemente se interpreta. Sin embargo, el método de *justo a tiempo* proporciona, de cualquier modo, una importante mejora en el desempeño. Aunque la compilación dinámica se aplica al código de bytes, las características de portabilidad y seguridad todavía aplicarán, pues el sistema en tiempo de ejecución (el cual realiza la compilación) estará aún a cargo del entorno de ejecución.

HABILIDAD FUNDAMENTAL

1.4 Terminología de Java

Ninguna revisión general de Java estaría completa sin que antes se diera un vistazo a la terminología de Java. Aunque las fuerzas fundamentales que se necesitaron para la invención de Java son la portabilidad y la seguridad, otros factores desempeñaron un papel importante en el modelado de la forma final del lenguaje. El equipo de diseño de Java resumió las consideraciones clave, están en la siguiente lista de términos.

Simple	Java tiene un conjunto conciso y cohesivo de funciones que facilitan su aprendizaje y uso.
Seguro	Java proporciona un medio seguro de crear aplicaciones de Internet.
Portable	Los programas de Java pueden ejecutarse en cualquier entorno para el cual haya un sistema de Java en tiempo de ejecución.
Orientado a objetos	Java encarna la filosofía moderna de programación orientada a objetos.
Robusto	Java alienta una programación libre de errores, pues requiere una escritura estricta y realizar comprobaciones en tiempo de ejecución.
Subprocesos múltiples	Java proporciona un soporte integrado para la programación de subprocesos múltiples.
Arquitectura neutra	Java no está unido a una máquina o una arquitectura específicas de sistema operativo.
Interpretado	Java soporta un código de plataforma cruzada mediante el uso de un código de bytes de Java.
Alto desempeño	El código de bytes de Java está altamente optimizado para que se ejecute rápidamente.
Distribuido	Java fue diseñado tomando en consideración el entorno distribuido de Internet.
Dinámico	Los programas de Java incluyen importantes cantidades de información que son del tipo de tiempo de ejecución. Esta información se usa para verificar y resolver el acceso a objetos al momento de realizar la ejecución.

Pregunte al experto

P: **Para atender los temas de la portabilidad y la seguridad, ¿por qué fue necesario crear un nuevo lenguaje de computación como Java?, ¿por qué no se adaptó un lenguaje como C++? En otras palabras, ¿no podría crearse un compilador de C++ que dé salida a un código de bytes?**

R: Aunque sería posible que un compilador de C++ generara un código de bytes en lugar de un código ejecutable, C++ tiene funciones que no recomiendan utilizarlo para la creación de applets; la más importante de ellas es el soporte a apuntadores de C++. Un *apuntador* es la dirección de algún objeto almacenado en la memoria. Con el uso de un apuntador, sería posible acceder a recursos fuera del propio programa, lo que daría como resultado una brecha de seguridad. Java no soporta apuntadores, con lo cual se elimina dicho problema.

Comprobación de avance

1. ¿Qué es un applet?
2. ¿Qué es el código de bytes de Java?
3. ¿Cuáles son los dos problemas que el uso de código de bytes ayuda a resolver?

HABILIDAD FUNDAMENTAL 1.5

Programación orientada a objetos

En el corazón de Java se encuentra la programación orientada a objetos (programación orientada a objetos, Object Oriented Programming). La metodología orientada a objetos es inseparable de Java, y todos los programas de Java son, hasta cierto punto, orientados a objetos. Debido a la importancia de la programación orientada a objetos, resulta útil comprender los principios básicos de ésta antes de escribir incluso el más simple programa de Java.

La programación orientada a objetos es una manera poderosa de afrontar el trabajo de programación. Las metodologías de programación han cambiado de manera importante desde la invención de la computadora, principalmente para adecuarse a la creciente complejidad de los programas. Por ejemplo, cuando se inventaron las computadoras, la programación se hacia al mover interruptores para ingresar las instrucciones binarias empleando el panel frontal de la computadora.

1. Un applet es un pequeño programa que se descarga dinámicamente de Web.
2. Un conjunto altamente optimizado de instrucciones que pueden ejecutarse en el intérprete de Java.
3. Portabilidad y seguridad.

Siempre y cuando los programas tuvieran unos cuantos cientos de instrucciones, este método funcionaba. A medida que los programas crecieron, se inventó el lenguaje ensamblador para que un programador pudiera tratar con programas más grandes y de complejidad creciente empleando representaciones simbólicas de las instrucciones de la máquina. A medida que los programas siguieron creciendo, se introdujeron lenguajes de nivel superior que proporcionaron al programador más herramientas con las cuales manejar la complejidad. El primer lenguaje de amplia difusión fue, por supuesto, FORTRAN. Aunque éste representó un primer paso impresionante, FORTRAN difícilmente es un lenguaje que estimula la creación de programas claros y fáciles de comprender.

La década de l960 vio nacer la programación estructurada. Es el método estimulado por lenguajes como C y Pascal. El uso de lenguajes estructurados permitió escribir con mucho mayor facilidad programas de complejidad moderada. Los lenguajes estructurados se caracterizan por su soporte de subrutinas independientes, variables locales y constructos de control, así como por su falta de dependencia de GOTO. Aunque los lenguajes estructurados constituyen una herramienta poderosa, alcanzan su límite cuando un proyecto se vuelve demasiado grande.

Tome en consideración lo siguiente: en cada momento clave del desarrollo de la programación, se crearon técnicas y herramientas que permitieron al programador tratar con una complejidad creciente. A cada paso, el nuevo método tomó los mejores elementos de los métodos anteriores y los hizo avanzar. Antes de la invención de la programación orientada a objetos, muchos proyectos estuvieron cerca del punto en que el método estructurado ya no funcionaba (o lo rebasaron). Los métodos orientados a objetos se crearon para ayudar a los programadores a rebasar estos límites.

La programación orientada a objetos retomó las mejores ideas de la programación estructurada y las combinó con varios conceptos nuevos. El resultado fue una nueva manera de organizar un programa. En el sentido más general, un programa puede organizarse mediante una de las siguientes dos maneras: alrededor de su código (lo que está sucediendo) o alrededor de sus datos (lo que se está afectando). Con el uso exclusivo de las técnicas de programación estructurada, los programas se encuentran típicamente organizados alrededor del código. A este método puede considerársele como "un código que actúa sobre los datos".

Los programas orientados a objetos funcionan de manera diferente: están organizados alrededor de los datos y el principio clave es que "los datos controlan el acceso al código". En un lenguaje orientado a objetos, usted define los datos y las rutinas a las que se les permite actuar sobre los datos. Por lo tanto, un tipo de datos define de manera precisa el tipo de operaciones que puede aplicarse a esos datos.

Para soportar los principios de la programación orientada a objetos, todos los lenguajes orientados a objetos, incluido Java, tienen tres rasgos en común: encapsulamiento, polimorfismo y herencia. Examinemos cada uno de ellos.

Encapsulamiento

El *encapsulamiento* es un mecanismo de programación que une al código y a los datos que manipula y que los mantiene a salvo de interferencias y de un mal uso externo. En un lenguaje orientado a objetos, el código y los datos pueden unirse de tal manera que pueda crearse una *caja negra* de contenido independiente. Dentro de la caja están todos los datos y el código necesarios. Cuando el código y los datos están vinculados de esta manera, se crea un objeto. En otras palabras, un objeto es el dispositivo que soporta el encapsulamiento.

Dentro de un objeto, el código, los datos, o ambos, pueden ser *privados*, o *públicos,* en relación con dicho objeto. El código o los datos privados son conocidos para la otra parte del objeto, y sólo ésta puede tener acceso a ellos. Es decir, una parte del programa que se encuentra fuera del objeto no puede acceder al código o los datos privados. Cuando el código o los datos son públicos, otras partes de su programa pueden acceder a ellos aunque estén definidos dentro del objeto. Por lo general, las partes públicas de un objeto se usan para proporcionar una interfaz controlada a los elementos privados de un objeto.

La unidad básica de encapsulamiento de Java es la *clase*. Si bien se examinarán las clases con mayor detalle en las páginas posteriores de este libro, el siguiente análisis breve le será de ayuda ahora. Una clase define la forma de un objeto; especifica los datos y el código que operarán sobre los datos. Java usa una especificación de clase para construir *objetos*. Los objetos son instancias de una clase. Por consiguiente, una clase es, en esencia, un conjunto de planos que especifican la manera de construir un objeto.

Al código y los datos que constituyen una clase se les denomina *miembros* de la clase. De manera específica, los datos definidos por la clase son denominados *variables de miembro* o *variables de instancia*. *Método* es el término que usa Java para una subrutina. Si está familiarizado con C/, C++, o ambos, le será de ayuda saber que lo que un programador de Java denomina *método*, un programador de C/C++ lo denomina *función*.

Polimorfismo

Polimorfismo (del griego "muchas formas") es la cualidad que permite que una interfaz acceda a una clase general de acciones. La acción específica está determinada por la naturaleza exacta de la situación. El volante de un automóvil representa un ejemplo simple de polimorfismo. El volante (es decir, la interfaz) es el mismo sin importar el tipo de mecanismo de conducción real que se emplee. En otras palabras, el volante funcionará de manera igual si su automóvil tiene dirección manual, dirección hidráulica o de engranes. Por lo tanto, una vez que sepa cómo operar el volante, podrá manejar cualquier tipo de automóvil.

El mismo principio se puede aplicar también a la programación. Por ejemplo, tome en consideración una pila (la cual es una lista del tipo primero en entrar y último en salir). Podría tener un programa que requiera tres tipos diferentes de pilas: una pila se usa para valores enteros, otra para valores de punto flotante y otra más para caracteres. En este caso, el algoritmo que implemente cada pila será el mismo, aunque los datos que se almacenen sean diferentes. En un lenguaje orientado a objetos necesitaría crear tres conjuntos diferentes de rutinas de pilas, y cada conjunto tendría que emplear nombres diferentes. Sin embargo, debido al polimorfismo, en Java puede crear un conjunto general de rutinas de pilas que funcione para las tres situaciones específicas. De esta manera, una vez que usted sabe cómo usar una pila, podrá usarlas todas.

De manera más general, el concepto de polimorfismo suele expresarse con la frase "una interfaz, varios métodos". Esto significa que es posible diseñar una interfaz genérica para un grupo de actividades relacionadas. El polimorfismo ayuda a reducir la complejidad al permitir que la misma interfaz sea usada para especificar una *clase general de acción*. Usted, el programador, no necesita llevar a cabo esta selección manualmente; sólo necesita recordar y utilizar la interfaz general.

Herencia

Herencia es el proceso mediante el cual un objeto puede adquirir las propiedades de otro objeto. Esto resulta importante porque soporta el concepto de clasificación jerárquica. En este sentido, la mayor parte del conocimiento se puede manejar mediante clasificaciones jerárquicas (es decir, de arriba a abajo). Por ejemplo, una manzana roja es parte de la clasificación *manzana*, que a su vez es parte de la clase *fruta*, la cual se encuentra bajo la clase más grande de *alimento*. Es decir, la clase *alimento* posee ciertas cualidades (comestible, nutritiva, etc.) que también aplican, lógicamente, a la subclase *fruta*. Además de estas cualidades, la clase *fruta* tiene características específicas (jugosa, dulce, etc.) que la distinguen de otros alimentos. La clase *manzana* define las cualidades específicas de una manzana (crece en árboles, no es tropical, etc.). Así, una manzana roja heredaría a su vez todas las cualidades de todas las clases anteriores y sólo definiría las cualidades que la hacen única.

Sin el uso de jerarquías, cada objeto tendría que definir explícitamente todas sus características. Si utiliza la herencia, un objeto sólo necesitará definir esas cualidades que lo hacen único dentro de su clase. De esta forma, el objeto puede heredar sus atributos generales a partir de su ascendiente y, por consiguiente, el mecanismo de la herencia hace posible que un objeto sea una instancia específica de un caso más general.

Comprobación de avance

1. Nombre los principios de la programación orientada a objetos.

2. ¿Cuál es la unidad básica de encapsulamiento en Java?

Pregunte al experto

P: **Usted estableció que la programación orientada a objetos es una manera efectiva de manejar programas largos. Sin embargo, al parecer dicha programación podría añadir una carga adicional a los programas pequeños. Debido a que usted mencionó que todos los programas de Java están, en cierta medida, orientados a objetos, ¿esto impone una penalidad a los programas más pequeños?**

R: No. Como verá, en el caso de programas pequeños, las funciones orientadas a objetos de Java son casi transparentes. Aunque es verdad que Java sigue un modelo estricto de objeto, usted cuenta con un amplio poder de decisión sobre el grado en el que lo emplea. Para el caso de programas más pequeños, sus características orientadas a objetos apenas son perceptibles. A medida que sus programas crezcan, usted integrará más características orientadas a objetos sin mayor esfuerzo.

1. Encapsulamiento, polimorfismo y herencia.
2. La clase.

Obtención del kit de desarrollo de Java

Ahora que se han explicado los pormenores teóricos de Java, es hora de empezar a escribir programas. Sin embargo, antes de que pueda compilar y ejecutar dichos programas, debe tener un sistema de desarrollo de Java instalado en su computadora. El que se emplea en este libro es el JDK (Java Development Kit, kit de desarrollo de Java) estándar, el cual está disponible en Sun Microsystems. Existen otros paquetes de desarrollo de otras compañías, pero usaremos el JDK porque está disponible para todos los lectores. Al momento de escribir este libro, la versión actual del JDK es la Java 2 Platform Standard Edition versión 5 (J2SE 5). Debido a que ésta contiene muchas características nuevas que no eran soportadas en versiones anteriores de Java, es necesario usar J2SE 5 (o posterior) para compilar y ejecutar los programas de este libro.

El JDK puede descargarse gratuitamente de **www.java.sun.com**. Sólo vaya a la página de descargas y siga las instrucciones para su tipo de computadora. Después de que haya instalado el JDK, estará listo para compilar y ejecutar programas. El JDK proporciona dos programas principales: el primero es **javac.exe**, que es el compilador de Java; el segundo es **java.exe**, que es el intérprete estándar de Java y también se denomina *lanzador de aplicaciones*.

Un comentario más: el JDK se ejecuta en el entorno del indicador de comandos o símbolo del sistema. No es una aplicación de entorno gráfico tipo Windows.

HABILIDAD FUNDAMENTAL 1.6

Un primer programa de ejemplo

Empecemos por compilar y ejecutar el programa corto de ejemplo que se muestra a continuación:

```
/*
   Éste es un programa simple de Java.

   Llame a este archivo Ejemplo.java.
*/
class Ejemplo {
  // Un programa de Java empieza con una llamada a main().
  public static void main(String args[]) {
    System.out.println("Java dirige Web.");
  }
}
```

Usted seguirá estos tres pasos:

1. Ingresar el programa.
2. Compilar el programa.
3. Ejecutar el programa.

Ingreso al programa

Los programas mostrados en este libro están disponibles en el sitio Web de Osborne: **www.osborne.com**. Sin embargo, si quiere ingresar los programas manualmente, tiene la libertad de hacerlo. En este caso, deberá ingresar el programa en su computadora empleando un editor de texto, no un procesador de palabras. Por lo general, los procesadores de Word almacenan información del formato junto con el texto. La información del formato confundirá al compilador de Java. Si está empleando una plataforma de Windows, puede usar WordPad o cualquier otro editor de programación que desee.

En el caso de la mayor parte de los lenguajes de computación, el nombre del archivo que contiene el código fuente de un programa es arbitrario. Sin embargo, éste no es el caso de Java. Lo primero que debe aprender acerca de Java es que *el nombre que asigne a un archivo fuente es muy importante*. Para este ejemplo, el nombre del archivo fuente debe ser **Ejemplo.java**. Veamos por qué.

En Java a un archivo fuente se le llama oficialmente *unidad de compilación*. Éste es un archivo de texto que contiene una o más definiciones de clase. El compilador de Java requiere que un archivo fuente use la extensión de nombre de archivo **.java**. Observe que la extensión de nombre de archivo tiene cuatro caracteres. Como bien podría suponer, su sistema operativo debe tener la capacidad de soportar nombres largos de archivo, lo cual significa que Windows 95, 98, NT, XP y 2000 funcionarán bien, pero no Windows 3.1.

Como verá al revisar el programa, el nombre de la clase definida por el programa también es **Ejemplo**. No se trata de una coincidencia. En Java, todo el código debe residir dentro de una clase. Por convención, el nombre de esa clase debe coincidir con el del archivo que contiene el programa. También debe asegurarse de que las mayúsculas y minúsculas del nombre de archivo coincidan con el nombre de la clase. La razón de ello es que Java es sensible a las mayúsculas y minúsculas. En este punto, es posible que la convención de que los nombres de archivo correspondan con los nombres de clase parezca arbitraria. Sin embargo, esta convención hace más fácil el mantenimiento y la organización de sus programas.

Compilación del programa

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

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

El compilador **javac** crea un archivo llamado **Ejemplo.class** que contiene la versión de código de bytes del programa. Recuerde que el código de bytes no es un código ejecutable. Debe ejecutarse en una Máquina Virtual de Java. Por lo tanto, la salida de **javac** no es un código que pueda ejecutarse directamente.

Para ejecutar realmente el programa, debe usar el intérprete de Java, es decir, **java**. Para ello, pase el nombre de clase **Ejemplo** como un argumento de línea de comandos, como se muestra a continuación:

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

Cuando el programa se ejecute, se desplegará la siguiente salida:

```
Java dirige Web.
```

Cuando el código fuente de Java se compila, cada clase individual se coloca en su propio archivo de salida llamado mediante el nombre de la clase y con la extensión **.class**. Por ello, resulta una buena idea asignar a su archivo fuente de Java el nombre de la clase que contiene: el nombre del archivo fuente coincidirá con el del archivo **.class**. Cuando ejecute el intérprete de Java como se acaba de mostrar, en realidad estará especificando el nombre de la clase que desee que el intérprete ejecute. Automáticamente buscará un archivo con ese nombre que tenga la extensión **.class**. Si encuentra el archivo, ejecutará el código que esté contenido en la clase especificada.

El primer programa de ejemplo, línea por línea

Aunque **Ejemplo.java** es muy corto, incluye varias características clave que le son comunes a todos los programas de Java. Examinemos de cerca cada parte del programa.

El programa empieza con las siguientes líneas:

```
/*
   Éste es un programa simple de Java.

   Llame a este archivo Ejemplo.java.
*/
```

Se trata de un *comentario*. Como casi todos los demás lenguajes de programación, Java le permite ingresar un comentario en el archivo fuente de un programa. El contenido de un comentario es ignorado por el compilador. En cambio, un comentario describe o explica la operación del programa a cualquier persona que esté leyendo su código fuente. En este caso, el comentario describe el programa y le recuerda que el archivo fuente debe llamarse **Ejemplo.java**. Por supuesto, en aplicaciones reales, los comentarios suelen explicar la manera en la que funciona alguna parte del programa, o bien, lo que una característica específica lleva a cabo.

Java soporta tres estilos de comentarios: el que se muestra en la parte superior del programa se llama *comentario de varias líneas*. Este tipo de comentario debe empezar con /* y terminar con */. Todo lo que se encuentre entre estos dos símbolos de comentario es ignorado por el compilador. Como el nombre lo sugiere, un comentario de varias líneas puede tener varias líneas de largo.

La siguiente línea del código del programa se muestra a continuación:

```
class Ejemplo {
```

Esta línea usa la palabra clave **class** para declarar que se está definiendo una nueva clase. Como ya se mencionó, la clase es la unidad básica de encapsulamiento de Java. **Ejemplo** es el nombre de la clase. La definición de clase empieza con una llave de apertura ({) y termina con una de cierre (}). Los elementos entre las dos llaves son miembros de la clase. Por el momento, no se preocupe demasiado por los detalles de una clase, pero tome en cuenta que en Java toda la actividad del programa ocurre dentro de una. Ésta es la razón de que los programadores de Java estén (por lo menos un poco) orientados a objetos.

La siguiente línea del programa es el *comentario de una sola línea, el cual* se muestra aquí:

```
// Un programa de Java empieza con una llamada a main().
```

Éste es el segundo tipo de comentario soportado por Java. Un *comentario de una sola línea* comienza con // y termina al final de la línea. Como regla general, los programadores usan comentarios de varias líneas para comentarios más largos y de una sola línea para descripciones breves, línea por línea.

A continuación se muestra la siguiente línea del código:

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

Esta línea empieza el método **main()**. Como ya se mencionó, en Java, a una subrutina se le llama *método*. Como se sugiere en el comentario anterior, ésta es la línea en la que el programa empezará a ejecutarse. Todas las aplicaciones de Java empiezan la ejecución mediante una llamada a **main()**. Por el momento, no puede proporcionarse el significado exacto de cada parte de esta línea porque ello incluye una comprensión detallada de varias funciones adicionales de Java. Sin embargo, debido a que muchos de los ejemplos de este libro usarán esta línea de código, echaremos a continuación un breve vistazo a cada parte.

La palabra clave **public** es un *especificador de acceso*. Un especificador de acceso determina la manera en la que otras partes de un programa pueden acceder a los miembros de la clase. Cuando un miembro de una clase está precedido por **public**, entonces es posible acceder a dicho miembro mediante un código que esté fuera de la clase en la que se encuentre declarado. (Lo opuesto de **public** es **private**, lo cual evita que un miembro sea utilizado por un código definido fuera de su clase.) En este caso, **main( )** debe declararse como **public** porque debe ser llamado por el código fuera de su clase cuando el programa se inicie. La palabra clave **static** permite que **main( )** sea llamado por el intérprete de Java antes de que se cree un objeto de la clase. Esto resulta necesario porque **main( )** es llamado por el intérprete de Java antes de que se haga cualquier objeto. La palabra clave **void** simplemente le indica al compilador que **main( )** no regresa un valor. Como verá, los métodos también pueden regresar valores. Si todo esto parece un poco confuso, no se preocupe. Todos estos conceptos se analizarán de manera detallada en módulos posteriores.

Como ya se estableció, **main( )** es el método al cual se llama al iniciar una aplicación de Java. Cualquier información que necesite pasar a un método es recibida por variables especificadas dentro del conjunto de paréntesis que viene después del nombre del método. A estas variables se les denomina *parámetros*. Si no se requieren parámetros para un método determinado, necesitará incluir de cualquier modo los paréntesis vacíos. En **main( )** sólo hay un parámetro, **String args[ ]**, el cual declara un parámetro denominado **args**. Se trata de una matriz de objetos del tipo **String**. (Las *matrices* son colecciones de objetos similares.) Los objetos de tipo **String** almacenan secuencias de caracteres. En este caso, **args** recibe cualquier argumento de línea de comandos que esté presente al momento de ejecutar el programa. Este programa no usa esta información; sin embargo, otros programas que se presentarán más adelante sí la utilizarán.

El último carácter de esta línea es la {. Esto señala el inicio del cuerpo de **main( )**. Todo el código incluido en un método ocurrirá entre la llave de apertura del método y su llave de cierre.

A continuación se muestra la siguiente línea de código. Note que esto ocurre dentro de **main( )**

```
System.out.println() ("Java drives the Web.");
```

Esta línea da salida a la cadena "Java dirige Web." seguida por una nueva línea en la pantalla. En realidad la salida se logra mediante el método integrado **println( )**. En este caso **println( )** despliega

la cadena que se le pasa. Como verá, **println( )** puede usarse también para desplegar otros tipos de información. La línea empieza con **System.out**. Aunque resulta demasiado complicada para explicarla ahora de manera detallada, en resumen, **System** es una clase predefinida que proporciona acceso al sistema y **out** es el flujo de salida que está conectado a la consola. Por consiguiente, **System.out** es un objeto que encapsula la salida de la consola. El hecho de que Java use un objeto para definir la salida de la consola constituye una evidencia adicional de que su naturaleza está orientada a objetos.

Como tal vez habrá adivinado, en los programas y los applets reales de Java no se emplea con frecuencia la salida (ni la entrada) de la consola. Debido a que la mayor parte de los entornos modernos de cómputo usan el paradigma de ventanas y tienen una naturaleza gráfica, la consola de E/S se emplea principalmente para utilerías simples y para programas de demostración. Más adelante aprenderá otras maneras de generar salida empleando Java, pero por ahora seguiremos usando los métodos de E/S de la consola.

Tome en cuenta que la instrucción **println( )** termina con un punto y coma. En Java, todas las instrucciones terminan con un punto y coma. La razón de que otras líneas del programa no terminen con punto y coma es que no son instrucciones en un sentido técnico.

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

Un último comentario: Java es sensible a las mayúsculas y minúsculas. Si lo olvida, se puede meter en problemas serios. Por ejemplo, si escribe por accidente **Main** en lugar de **main**, o **PrintLn** en lugar de **println**, el programa anterior será incorrecto. Más aún, si bien el compilador de Java *compilará* clases que no contengan un método main(), no tendrá manera de ejecutarlas. De este modo, si escribe **main** de manera incorrecta, el compilador compilará de cualquier modo su programa. Sin embargo, el intérprete de Java reportará un error porque no encontrará el método **main( )**.

Comprobación de avance

1. ¿Dónde empieza la ejecución de un programa de Java?
2. ¿Qué hace **System.out.println( )**?
3. ¿Cuáles son los nombres del compilador y el intérprete de Java?

1. **main( )**
2. Da salida a la información en la consola.
3. El compilador estándar de Java es **javac.exe**, el intérprete es **java.exe**.

Manejo de errores de sintaxis

Si aún no lo ha hecho, ingrese, compile y ejecute el programa anterior. Como ya lo sabrá por su experiencia en programación, es muy fácil escribir por accidente algo incorrecto al momento de ingresar código en su computadora. Por fortuna, si ingresa algo de manera incorrecta en su programa, el compilador reportará un mensaje de *error de sintaxis* cuando trate de compilarlo. El compilador de Java trata de darle sentido a su código fuente, sin importar lo que haya escrito. Por tal motivo, el error que se reporte tal vez no refleje la causa real del problema. En el programa anterior, por ejemplo, una omisión accidental de la llave de apertura después del método **main( )** causará que el compilador reporte la siguiente secuencia de errores.

```
Ejemplo.java:8: ';' expected
  Public static void main(String args[])
                                        ^
Ejemplo.java:11: 'class' or 'interface' expected
}
^
Ejemplo.java:13: 'class' or 'interface' expected
^
Ejemplo.java:8: missing method body, or declare abstract
  Public static void main(String args[])
```

Es claro que el primer mensaje de error es totalmente incorrecto, pues lo que hace falta no es el punto y coma sino una llave.

Lo importante de este análisis es que cuando su programa contenga un error de sintaxis, no necesariamente deberá tomar al pie de la letra los mensajes del compilador, ya que éstos pueden ser incorrectos. Así que, necesitará realizar una "segunda suposición" a partir de un mensaje de error para encontrar el problema real. Además, revise en su programa las últimas líneas de código que antecedan a la línea que se está marcando. En ocasiones no se reportará un error sino hasta varias líneas después de que el error se haya realmente presentado.

HABILIDAD FUNDAMENTAL 1.7

Un segundo programa simple

Tal vez ninguna otra construcción sea tan importante en un lenguaje de programación como la asignación de un valor a una variable. Una *variable* es una ubicación de memoria con un nombre a la cual se le puede asignar un valor. Más aún, el valor de una variable puede cambiar durante la ejecución de un programa. Es decir, el contenido de una variable es modificable, no fijo.

El siguiente programa crea dos variables llamadas **var1** y **var2**.

```
/*
   Esto demuestra una variable.

   Llame a este archivo Ejemplo2.java.
*/
class Ejemplo2 {
  public static void main(String args[]) {
    int var1; // esto declara una variable
    int var2; // esto declara otra variable

    var1 = 1024; // esto asigna 1024 a var1

    System.out.println("var1 contiene " + var1);


    var2 = var1 / 2;


    System.out.print("var2 contiene var1 / 2: ");
    System.out.println(var2);
  }
}
```

Declara variables.

Asigna un valor a una variable.

Cuando ejecute este programa, verá la siguiente salida:

```
var1 contiene 1024
var2 contiene var1 / 2: 512
```

Este programa introduce varios conceptos nuevos. Primero, la instrucción

```
int var1; // esto declara una variable
```

declara una variable llamada **var1** de tipo entero. En Java, todas las variables deben declararse antes de usarse. Más aún, debe también especificarse el tipo de valores que la variable puede contener. A esto se le denomina *tipo* de variable. En este caso, **var1** puede contener valores enteros. En Java, para declarar que una variable es entera, su nombre debe estar antecedido por la palabra clave **int**. Por lo tanto, la instrucción anterior declara una variable llamada **var1** del tipo **int**.

La siguiente línea declara una segunda variable denominada **var2**.

```
int var2; // esto declara otra variable
```

Observe que esta línea usa el mismo formato que la primera, con excepción de que el nombre de la variable es diferente.

En general, para declarar una variable tendrá que usar una instrucción como ésta:

tipo nombre-var;

En este caso, *tipo* especifica el tipo de variable que se está declarando y *nombre-var* es el nombre de la variable. Además de **int**, Java soporta otros tipos de datos.

La siguiente línea de código asigna a **var1** el valor 1024:

```
var1 = 1024; // esto asigna 1024 a var1
```

En Java, el operador de asignación es un solo signo de igual. Copia el valor de la derecha en la variable de la izquierda.

La siguiente línea de código da salida al valor de **var1** antecedido por la cadena "var1 contiene ":

```
System.out.println("var1 contiene " + var1);
```

En esta instrucción, el signo de más hace que el valor de **var1** se despliegue después de la cadena que lo antecede. Es posible generalizar este método. Con el uso del operador +, puede unir en una cadena todos los elementos que desee dentro de una sola instrucción **println()**.

La siguiente línea de código asigna a **var2** el valor de **var1** dividido entre 2:

```
var2 = var1 / 2;
```

Esta línea divide el valor de **var1** entre 2 y luego almacena ese resultado en **var2**. Por lo tanto, después de ejecutar la línea, **var2** contendrá el valor 512. El valor de **var1** permanecerá sin cambio. Como casi todos los demás lenguajes de cómputo, Java soporta un rango completo de operadores aritméticos, incluidos los que se muestran a continuación:

+	Suma
-	Resta
*	Multiplicación
/	División

He aquí las dos líneas siguientes del programa:

```
System.out.print("var2 contiene var1 / 2: ");
System.out.println(var2);
```

Dos cosas nuevas ocurren en este último caso. En primer lugar, se usa el método integrado **print()** para desplegar la cadena "var2 contiene var1 / 2: ". Esta cadena *no* es seguida por una línea nueva. Esto significa que cuando la siguiente salida se genere, ésta empezará en la misma línea. El método **print()** es como **println()**, a excepción de que no da salida a una nueva línea después de cada llamada.

En segundo lugar, en la llamada a **println()**, observe que se usa **var2** por sí sola. Tanto **print()** como **println()** pueden usarse para dar salida a valores de cualquiera de los tipos integrados de Java.

Antes de seguir adelante, un comentario más acerca de la declaración de variables: es posible declarar dos o más variables empleando la misma instrucción de declaración. Tan sólo separe sus nombres mediante comas. Por ejemplo, pudo declarar **var1** y **var2** de esta manera:

```
int var1, var2; // ambas se declaran usando una instrucción
```

Otro tipo de datos

En el ejemplo anterior se usó una variable del tipo **int**. Sin embargo, este tipo de variable sólo puede contener números enteros. Por lo tanto, no puede usarse cuando se requiera un componente fraccionario. Por ejemplo, una variable **int** puede contener el valor 18, pero no 18.3. Por fortuna, **int** sólo es uno de los varios tipos de datos definidos por Java. Para permitir números con componentes fraccionarios, Java define dos tipos de punto flotante: **float** y **double**, los cuales representan valores de precisión sencilla y doble, respectivamente. De los dos, **double** es de uso más común.

Para declarar una variable del tipo **double**, utilice una instrucción similar a la que se muestra a continuación:

```
double x;
```

Aquí, **x** es el nombre de la variable, la cual es de tipo **double**. Debido a que **x** tiene un tipo de punto flotante, puede contener valores como 122.23, 0.034, o –19.0.

Para comprender mejor las diferencias entre **int** y **double**, pruebe el siguiente programa:

```
/*
   Este programa ilustra las diferencias
   entre int y double.

   Llame a este archivo Ejemplo3.java.
*/
class Ejemplo3 {
  public static void main(String args[]) {
    int var;  // esto declara una variable int
    double x; // esto declara una variable de punto flotante

    var = 10; // asigna a var el valor 10

    x = 10.0; // asigna a x el valor 10.0

    System.out.println("Valor original de var: " + var);
    System.out.println("Valor original de x: " + x);
```

```
    System.out.println(); // imprime una línea en blanco
    // ahora divide ambos entre 4
    var = var / 4;
    x = x / 4;

    System.out.println("var una vez dividida: " + var);
    System.out.println("x una vez dividida: " + x);
  }
}
```

Imprime una línea en blanco.

Aquí se muestra la salida de este programa:

```
Valor original de var: 10
Valor original de x: 10.0

var una vez dividida: 2
x una vez dividida: 2.5
```

Componente fraccionario perdido

Componente fraccionario preservado

Como puede observar, cuando **var** se divide entre 4, se realiza una división entre enteros, y el resultado es 2 (el componente fraccionario se pierde). Sin embargo, cuando **x** se divide entre 4, el componente fraccionario se conserva y se despliega la respuesta apropiada.

Hay algo más que es posible observar en el programa: para imprimir una línea en blanco, simplemente llame a **println()** sin ningún tipo de argumentos.

Pregunte al experto

P: **¿Por qué Java tiene diferentes tipos de datos para valores enteros y de punto flotante? Es decir, ¿por qué no todos los valores numéricos son del mismo tipo?**

R: Java proporciona diferentes tipos de datos para que pueda escribir programas eficientes. Por ejemplo, la aritmética de los enteros es más rápida que la de los cálculos de punto flotante. Por lo tanto, si no necesita valores fraccionarios, no es necesario que incurra en la carga adicional asociada con los tipos **float** y **double**. En segundo lugar, la cantidad de memoria requerida para un tipo de datos podría ser menor que la necesaria para otro. Al proporcionar diferentes tipos, Java le permite utilizar mejor los recursos del sistema. Por último, algunos algoritmos requieren el uso de un tipo específico de datos (o por lo menos se benefician de él). En general, Java proporciona varios tipos integrados para ofrecerle una mayor flexibilidad en el procesador Pentium de Intel.

Proyecto 1.1 Conversión de galones a litros

GalALit.java

Aunque ilustran varias características importantes del lenguaje de Java, los programas anteriores de ejemplo no resultan muy útiles. A pesar de que en este momento todavía no sabe mucho acerca de Java, puede poner en práctica lo que ha aprendido para crear un programa práctico. En este proyecto, crearemos un programa que convierta galones en litros.

El programa funcionará al declarar dos variables **double**. Una contendrá el número de galones y la segunda contendrá el número de litros tras la conversión. Hay 3.7854 litros en un galón. Por lo tanto, para convertir galones en litros, el valor del galón se multiplica por 3.7854. El programa despliega el número de galones y el número equivalente de litros.

Paso a paso

1. Cree un nuevo archivo llamado **GalALit.java**.
2. Ingrese el siguiente programa en el archivo:

```
/*
   Proyecto 1.1

   Este programa convierte galones en litros.

   Llame a este programa GalALit.java.
*/
class GalALit {
  public static void main(String args[]) {
    double galones; // contiene la cantidad de galones
    double litros;  // contiene lo convertido a litros

    galones = 10; // empieza con 10 galones

    litros = galones * 3.7854; // convierte a litros

    System.out.println(galones + " galones son " + litros + " litros.");
  }
}
```

3. Compile el programa empleando la siguiente línea de comandos:

```
C>javac GalALit.java
```

4. Ejecute el programa empleando este comando:

```
C>java GalALit
```

Verá esta salida:

```
10.0 galones son 37.854 litros.
```

5. En el estado actual, este programa convierte 10 galones en litros. Sin embargo, al cambiar el valor asignado a **galones**, puede hacer que el programa convierta un número diferente de galones en su número equivalente de litros.

Comprobación de avance

1. ¿Cuál es la palabra clave de Java para el tipo de datos enteros?
2. ¿Qué es **double**?

HABILIDAD FUNDAMENTAL 1.8

Dos instrucciones de control

Dentro de un método, la ejecución pasa de una instrucción a la siguiente, de arriba a abajo. Sin embargo, es posible modificar este flujo mediante el uso de varias instrucciones de control de programa soportadas por Java. Aunque más adelante revisaremos detenidamente las instrucciones de control, a continuación se introducirán brevemente dos pues son las que estaremos empleando para escribir programas de ejemplo.

La instrucción if

Puede ejecutar selectivamente parte de un programa mediante el uso de la instrucción condicional de Java: **if**. La instrucción **if** de Java funciona de manera muy parecida a la instrucción IF de otros lenguajes. Aquí se muestra su forma más simple:

if(*condición*) *instrucción*;

En este caso, *condición* es una expresión Booleana. Si la *condición* es verdadera, entonces se ejecuta la instrucción. Si la *condición* es falsa, entonces se omite la instrucción. He aquí un ejemplo:

```
if(10 < 11) System.out.println("10 es menorque 11");
```

En este caso, debido a que 10 es menor que 11, la expresión condicional es verdadera, y **println()** se ejecutará. Sin embargo, considere lo siguiente:

```
if(10 < 9) System.out.println("esto no se muestra");
```

En este caso, 10 no es menor que 9, por lo que la llamada a **println()** no tendrá lugar.

1. **int**
2. La palabra clave para el tipo de datos de punto flotante **double.**

Java define un complemento completo de operadores relacionales que pueden usarse en una expresión condicional. Se muestran aquí:

Operador	Significado
<	Menor que
<=	Menor que o igual a
>	Mayor que
>=	Mayor que o igual a
==	Igual a
!=	No igual

Observe que la prueba de igualdad es el doble signo de igual.

He aquí un programa que ilustra la instrucción **if**:

```
/*
  Demuestra if.

  Llame a este archivo IfDemo.java.
*/
class IfDemo {
  public static void main(String args[]) {
    int a, b, c;

    a = 2;
    b = 3;

    if(a < b) System.out.println("a es menor que b");

    // esto no despliega nada
    if(a == b) System.out.println("no va a ver esto");

    System.out.println();

    c = a - b; // c contiene -1

    System.out.println("c contiene -1");
    if(c >= 0) System.out.println("c no es negativo");
    if(c < 0) System.out.println("c es negativo");

    System.out.println();

    c = b - a; // c contiene ahora 1
```

```
    System.out.println("c contiene 1");
    if(c >= 0) System.out.println("c no es negativo");
    if(c < 0) System.out.println("c es negativo");

  }
}
```

Aquí se muestra la salida generada por este programa:

```
a es menor que b

c contiene -1
c es negativo

c contiene 1
c no es negativo
```

Observe otro aspecto en este programa. La línea

```
int a, b, c;
```

declara tres variables, **a**, **b** y **c**, mediante el uso de una lista separada por comas. Como se mencionó antes, si necesita dos o más variables del mismo tipo, éstas pueden declararse en una instrucción. Sólo separe los nombres de las variables con comas.

El bucle for

Puede ejecutar de manera repetida una secuencia de código al crear un *bucle*. Java proporciona una diversidad enorme de constructos de bucle. El que observaremos aquí es el bucle **for**. La forma más simple de éste se muestra a continuación:

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

En su forma más común, la parte de *inicialización* del bucle asigna un valor inicial a una variable de control de bucle. La *condición* es una expresión booleana que prueba la variable de control de bucle. Si el resultado de esa prueba es verdadero, el bucle **for** sigue repitiéndose. Si es falsa, el bucle se termina. La expresión *iteración* determina la manera en la que la variable de control de bucle cambia cada vez que el bucle se repite. He aquí un programa corto que ilustra el bucle **for**:

```
/*
  Demuestra el bucle for.

  Llame a este archivo ForDemo.java.
```

```
*/
class ForDemo {
  public static void main(String args[]) {
    int cuenta;

    for(cuenta = 0; cuenta < 5; cuenta = cuenta+1) ← Éste bucle se repite cinco veces.
      System.out.println("Esta es la cuenta: " + cuenta);

    System.out.println("Fin");
  }
}
```

La salida generada por el programa se muestra aquí:

```
Ésta es la cuenta: 0
Ésta es la cuenta: 1
Ésta es la cuenta: 2
Ésta es la cuenta: 3
Ésta es la cuenta: 4
Fin
```

En este ejemplo, **cuenta** es la variable de control del bucle. Se establece en cero en la parte de inicialización de **for**. Al principio de cada iteración (incluida la primera), se realiza la prueba de condición **cuenta < 5**. Si la salida de esta prueba es verdadera, la instrucción **println()** se ejecuta, y luego se ejecuta la parte de la iteración del bucle. Este proceso sigue hasta que la prueba de la condición resulta falsa, momento en el que la ejecución pasa a la parte inferior del bucle.

Como punto de interés, en los programas de Java escritos profesionalmente, casi nunca verá la parte de la iteración del bucle escrita como en el programa anterior; es decir, casi nunca verá instrucciones como ésta:

```
cuenta = cuenta + 1;
```

La razón de ello es que Java incluye un operador especial de incremento que realiza esta operación de manera más eficiente. El operador de incremento es ++ (es decir, dos signos de más, uno tras otro). El operador aumenta su operando en uno. Mediante el uso del operador de incremento, la instrucción anterior se escribiría así:

```
cuenta++;
```

Por lo tanto, el **for** del programa anterior se escribiría normalmente de la manera siguiente:

```
for(cuenta = 0; cuenta < 5; cuenta++)
```

Tal vez quiera probar esto. Como verá, el bucle sigue ejecutándose tal y como lo hizo antes.

Java también proporciona un operador de decremento, el cual se especifica como – –. Este operador disminuye su operando en uno.

Comprobación de avance

1. ¿Qué hace la instrucción **if**?
2. ¿Qué hace la instrucción **for**?
3. ¿Cuáles son los operadores relacionales de Java?

HABILIDAD FUNDAMENTAL 1.9

Cree bloques de código

Otro elemento clave de Java es el *bloque de código*. Un bloque de código es la agrupación de dos o más instrucciones, lo cual se lleva a cabo al encerrar las instrucciones entre llaves de apertura y cierre. Una vez que se ha creado un bloque de código, éste se vuelve una unidad lógica que puede usarse en cualquier lugar en el que podría usarse una sola instrucción. Por ejemplo, un bloque puede ser un destino para instrucciones **if** y **for** de Java. Revise la siguiente instrucción **if**:

```
if(w < h) {   <-- Inicio del bloque
  v = w * h;
  w = 0;
}             <-- Fin del bloque
```

Aquí, si **w** es menor que **h**, se ejecutarán ambas instrucciones dentro del bloque. Por consiguiente, las dos instrucciones dentro del bloque forman una unidad lógica, así que una instrucción no puede ejecutarse sin que la otra también se ejecute. La clave es que cada vez que necesite vincular lógicamente dos o más instrucciones, lo hará si crea un bloque. Los bloques de código permiten que muchos algoritmos se implementen con mayor claridad y eficiencia.

1. **if** es la instrucción de condición de Java.
2. **for** es una de las instrucciones de bucle de Java.
3. Los operadores relacionales son ==, !=, <, >, <= y >=.

He aquí un programa que usa un bloque de código para evitar una división entre cero:

```
/*
  Demuestra un bloque de código.

  Llame a este archivo BloqueDemo.java.
*/
class BloqueDemo {
  public static void main(String args[]) {
    double i, j, d;

    i = 5;
    j = 10;

    // el destino de este if es un bloque
    if(i != 0) {
      System.out.println("i no es igual a cero");
      d = j / i;
      System.out.print("j / i es " + d);
    }
  }
}
```

El destino del **if** es todo el bloque.

Aquí se muestra la salida generada por este programa:

```
i no es igual a cero
j / i es 2.0
```

En este caso, el destino de la instrucción **if** es un bloque de código y no una sola instrucción. Si la condición que controla el **if** es verdadera (como en este caso), entonces se ejecutarán las tres instrucciones dentro del bloque. Haga la prueba asignando cero a **i** y observe el resultado.

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

Pregunte al experto

P: **¿El uso de un bloque de código introduce alguna ineficiencia en tiempo de ejecución? En otras palabras, ¿en realidad Java ejecuta { y }?**

R: No. Los bloques de código no agregan ninguna carga adicional. En realidad, debido a su capacidad de simplificar la codificación de ciertos algoritmos, su uso generalmente aumenta la velocidad y la eficiencia. Además, { y } sólo existen en el código fuente de su programa. En sí, Java no ejecuta { o }.

HABILIDAD FUNDAMENTAL 1.10

Punto y coma y posicionamiento

En Java, el punto y coma es *el que finaliza* una instrucción. Es decir, cada instrucción individual debe terminar con un punto y coma pues indica el final de una entidad lógica.

Como ya lo sabe, un bloque es un conjunto de instrucciones conectadas lógicamente y que están encerradas entre llaves de apertura y cierre. Un bloque *no* termina con un punto y coma. Como un bloque es un grupo de instrucciones, con un punto y coma después de cada instrucción, es lógico entonces que éste no termine con un punto y coma. En cambio, el final del bloque está indicado por la llave de cierre.

Java no reconoce el final de la línea como terminación. Por ello, no importa en qué parte de la línea ponga su instrucción. Por ejemplo,

```
x = y;
y = y + 1;
System.out.println(x + " " + y);
```

Para Java, es lo mismo que lo siguiente:

```
x = y;  y = y + 1;  System.out.println(x + " " + y);
```

Más aún, los elementos individuales de una instrucción también pueden ponerse en líneas separadas. Por ejemplo, lo siguiente es perfectamente aceptable.

```
System.out.println("Ésta es una línea larga de salida" +
                   x + y + z +
                   "más salida");
```

Esta manera de dividir líneas largas se utiliza para que los programas sean más legibles. También puede ayudar a evitar que líneas excesivamente largas se dividan al final de la línea.

Prácticas de sangrado

Tal vez haya notado en los ejemplos anteriores que ciertas instrucciones están sangradas. Java es un lenguaje de forma libre, lo que significa que no importa dónde coloque las instrucciones en relación con las demás instrucciones de una línea. Sin embargo, con los años se ha desarrollado y aceptado un estilo de sangrado común que permite la creación de programas muy legibles. En este libro se sigue este estilo, por lo que se recomienda que también lo utilice. Con este estilo, usted sangra un nivel después de cada llave de apertura y regresa un nivel hacia el margen después de cada llave de cierre. Ciertas instrucciones estimulan un sangrado adicional, lo cual se analizará más adelante.

Comprobación de avance

1. ¿Cómo se crea un bloque de código? ¿Qué es lo que hace?
2. En Java las instrucciones terminan con un _____.
3. Todas las instrucciones de Java deben empezar y terminar en una línea. ¿Cierto o falso?

Proyecto 1.2 Mejoramiento del convertidor de galones en litros

GalALitTabla.java

Puede usar el bucle **for**, la instrucción **if** y los bloques de código para crear una versión mejorada del convertidor de galones a litros que desarrolló en el primer proyecto. Esta nueva versión imprimirá una tabla de conversiones que empieza con 1 galón y termina con 100 galones. Después de cada 10 galones, se dará salida a una línea en blanco. Esto se realiza con el uso de una variable llamada **contador** que cuenta el número de líneas que se han enviado a la salida. Ponga especial atención en su uso.

Paso a paso

1. Cree un nuevo archivo llamado **GalALitTabla.java.**
2. Ingrese el siguiente programa en el archivo.

```
/*
   Proyecto 1.2

   Este programa despliega una tabla para
   convertir galones en litros.

   Llame a este programa "GalALitTabla.java".
*/
class GalALitTabla {
  public static void main(String args[]) {
    double galones, litros;
    int contador;

    contador = 0;
    for(galones = 1; galones <= 100; galones++) {
      litros = galones * 3.7854; // convierte a litros
```

El contador de línea está inicialmente en cero.

1. Un bloque inicia con una {. Termina con una }. Un bloque crea una unidad lógica de código.
2. Punto y coma.
3. Falso.

```
      System.out.println(galones + " galones son " +
                         litros + " litros.");

      contador++;
      // cada 10 líneas, imprime una línea en blanco
      if(contador == 10) {
        System.out.println();
        contador = 0; // restablece el contador de líneas
      }
    }
  }
}
```

Incrementa el contador de líneas con cada iteración del bucle.

Si el contador llega a 10, se da salida a una línea en blanco.

3. Compile el programa empleando la siguiente línea de comandos:

```
C>javac GalALitTabla.java
```

4. Ejecute el programa empleando este comando:

```
C>java GalALitTabla
```

He aquí una parte de la salida que verá:

```
1.0 galones son 3.7854 litros.
2.0 galones son 7.5708 litros.
3.0 galones son 11.356200000000001 litros.
4.0 galones son 15.1416 litros.
5.0 galones son 18.927 litros.
6.0 galones son 22.712400000000002 litros.
7.0 galones son 26.4978 litros.
8.0 galones son 30.2832 litros.
9.0 galones son 34.0686 litros.
10.0 galones son 37.854 litros.

11.0 galones son 41.6394 litros.
12.0 galones son 45.424800000000005 litros.
13.0 galones son 49.2102 litros.
14.0 galones son 52.9956 litros.
15.0 galones son 56.781 litros.
16.0 galones son 60.5664 litros.
17.0 galones son 64.3518 litros.
18.0 galones son 68.1372 litros.
19.0 galones son 71.9226 litros.
20.0 galones son 75.708 litros.

21.0 galones son 79.49340000000001 litros.
22.0 galones son 83.2788 litros.
23.0 galones son 87.0642 litros.
24.0 galones son 90.84960000000001 litros.
25.0 galones son 94.635 litros.
```

(continúa)

```
26.0 galones son 98.4204 litros.
27.0 galones son 102.2058 litros.
28.0 galones son 105.9912 litros.
29.0 galones son 109.7766 litros.
30.0 galones son 113.562 litros.
```

HABILIDAD FUNDAMENTAL 1.11

Las palabras clave de Java

Actualmente están definidas 50 palabras clave en el lenguaje Java (tabla 1.1). Estas palabras clave, combinadas con la sintaxis de los operadores y los separadores, forman la definición del lenguaje. Estas palabras clave no pueden usarse como nombres de variables, clases o métodos.

Las palabras clave **const** y **goto** están reservadas pero no se usan. En los primeros días de Java, se reservaron otras palabras clave para un posible uso futuro. Sin embargo, la especificación actual de Java sólo define las palabras clave mostradas en la tabla 1.1.

La palabra clave **enum** es muy reciente. Se añadió en J2SE 5. Además de las palabras clave, Java reserva las siguientes: **true**, **false** y **null**. Se trata de valores definidos por Java. No puede usar estas palabras como nombres de variables, clases, etc.

HABILIDAD FUNDAMENTAL 1.12

Identificadores en Java

En Java, un identificador es un nombre dado a un método, una variable o a cualquier otro elemento definido por el usuario. Los identificadores pueden tener uno o varios caracteres de largo. Los nombres de variables pueden empezar con cualquier letra del alfabeto, un guión de subrayado o un signo de pesos. Luego puede ser cualquier letra, un dígito, un signo de pesos o un guión de subrayado. Este guión puede usarse para mejorar la legibilidad del nombre de una variable, como en **cuenta_variables**.

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

Tabla 1.1 Las palabras clave de Java

Las mayúsculas y minúsculas son diferentes; es decir, para Java **mivar** y **MiVar** son nombres diferentes. He aquí algunos ejemplos de identificadores aceptables:

Prueba	x	y2	CargaMax
$up	_(bajo)*arriba*	*mi_(bajo)*var	muestra23

Recuerde que no puede iniciar un identificador con un dígito. Así que, por ejemplo, **12x** no es válido..

No puede utilizar ninguna de las palabras clave de Java como nombres de identificadores. Además, no debe asignar el nombre de ningún método estándar, como **println**, a un identificador. Más allá de estas dos restricciones, la buenas prácticas de programación dictan que emplee nombres de identificadores que reflejen el significado o el uso de los elementos a los que se les asigna el nombre.

Comprobación de avance

1. ¿Cuál es la palabra clave: **for**, **For** o **FOR**?
2. ¿Qué tipo de caracteres puede contener un identificador de Java?
3. ¿**Muestra21** y **muestra21** son el mismo identificador?

Las bibliotecas de clases de Java

Los programas de ejemplo que se mostraron en este módulo usan dos de los métodos integrados de Java: **println()** y **print()**. Estos métodos son miembros de la clase **System**, la cual es una clase predefinida de Java que se incluye automáticamente en sus programas. De manera general, el entorno de Java depende de varias bibliotecas de clases integradas que contienen muchos métodos integrados que proporcionan soporte para cosas como E/S, controladores de cadenas, red e imágenes. Las clases estándar también proporcionan soporte para una salida tipo Windows. Por lo tanto, Java es, en su totalidad, una combinación del propio lenguaje Java, más sus clases estándar. Como verá, las bibliotecas de clases proporcionan gran parte de las funciones que vienen con Java. Por supuesto, parte de la tarea de convertirse en un programador de Java consiste en aprender a usar las clases estándar de Java. A lo largo de este libro, se describirán varios elementos de las clases y métodos de la biblioteca estándar. Sin embargo, usted deseará explorar más ampliamente y por su cuenta la biblioteca de Java.

1. La palabra clave es **for**. en Java, todas las palabras clave se encuentran en minúsculas.
2. Letras, dígitos, el guión de subrayado y el signo de pesos.
3. No; Java es sensible a las mayúsculas y minúsculas.

Comprobación de dominio del módulo 1

1. ¿Qué es un código de bytes y por qué es importante su uso en Java para la programación de Internet?
2. ¿Cuáles son los tres principios de la programación orientada a objetos?
3. ¿Dónde empieza la ejecución de los programas de Java?
4. ¿Qué es una variable?
5. ¿Cuál de los siguientes nombres de variables no es válido?
   a) cuenta
   b) $cuenta
   c) cuenta 27
   d) 67cuenta
6. ¿Cómo crea un comentario de una sola línea? ¿Cómo crea uno de varias líneas?
7. Muestre la forma general de la instrucción **if**. Muestre la forma general del bucle **for**.
8. ¿Cómo se crea un bloque de código?
9. La gravedad de la Luna es de alrededor de 17% la de la Tierra. Escriba un programa que calcule su peso efectivo en la Luna.
10. Adapte el proyecto 1.2 para que imprima una tabla de conversión de pulgadas a metros. Despliegue 12 pies de conversiones, pulgada por pulgada. Dé salida a una línea en blanco cada 12 pulgadas. (Un metro es aproximadamente igual a 39.37 pulgadas.)
11. Si comete un error de escritura cuando esté ingresando su programa, ¿qué tipo de error aparecerá?
12. ¿Es importante el lugar de la línea en el que coloca una instrucción?

Módulo 2

Introducción a los tipos de datos y los operadores

HABILIDADES FUNDAMENTALES

2.1	Conozca los tipos primitivos de Java
2.2	Use literales
2.3	Inicialice variables
2.4	Conozca el alcance de las reglas de variables dentro de un método
2.5	Use los operadores aritméticos
2.6	Use los operadores relacionales y lógicos
2.7	Comprenda los operadores de asignación
2.8	Use asignaciones de métodos abreviados
2.9	Comprenda la conversión de tipos en asignaciones
2.10	Moldee tipos incompatibles
2.11	Comprenda la conversión de tipos en expresiones

En la base de cualquier lenguaje de programación se encuentran sus tipos de datos y sus operadores: Java no es la excepción. Estos elementos definen los límites de un lenguaje y determinan el tipo de tareas a las que pueden aplicarse. Por fortuna, Java soporta una rica variedad de tipos de datos y de operadores, lo que lo hace adecuado para cualquier tipo de programación.

Los tipos de datos y los operadores representan un tema extenso. Empezaremos aquí con un examen de los tipos de datos que son la base de Java y de sus operadores de uso más común. También echaremos un vistazo más de cerca a las variables y examinaremos la expresión.

¿Por qué los tipos de datos son importantes?

Los tipos de datos son especialmente importantes en Java porque es un lenguaje que requiere mucha escritura. Esto significa que el compilador revisa la compatibilidad de los tipos de todas las operaciones. Las operaciones ilegales no se compilarán. Por consiguiente, una revisión detallada de los tipos contribuye a evitar errores y a mejorar la confiabilidad. Para permitir una revisión tal de los tipos, todas las variables, expresiones y valores tienen un tipo. Por ejemplo, no existe el concepto de variable "sin tipo". Más aún, el tipo de un valor determina las operaciones que se permiten en él. Una operación permitida en un tipo tal vez no esté permitida en otro.

HABILIDAD FUNDAMENTAL 2.1

Tipos primitivos de Java

Java contiene dos categorías generales de tipos de datos integrados: orientados a objetos y no orientados a objetos. Los tipos orientados a objetos de Java están definidos por clases (el análisis de las clases se postergará para después). Sin embargo, en el corazón de Java hay ocho tipos de datos primitivos (también llamados elementales o simples), que se muestran en la tabla 2.1. El término *primitivo* se usa aquí para indicar que estos tipos no son objetos en el sentido de que están orientados a objetos, sino más bien valores binarios normales. Estos tipos primitivos no son objetos en cuanto a la eficiencia. Todos los otros tipos de datos de Java están construidos a partir de estos tipos primitivos.

Java especifica estrictamente un rango y un comportamiento para cada tipo primitivo, lo cual todas las implementaciones de la máquina virtual de Java deben soportar. Por ejemplo, un **int** es lo mismo en todos entornos de ejecución. Esto permite que los programas sean completamente portables. No es necesario reescribir un código para adecuarlo a una plataforma. Aunque la especificación estricta del tamaño de los tipos primitivos puede causar una pequeña pérdida de desempeño en algunos entornos, resulta necesaria para lograr la portabilidad.

Tipo	Significado
boolean	Representa valores verdaderos/falsos
byte	Entero de 8 bits
char	Carácter
double	Punto flotante de doble precisión
float	Punto flotante de precisión sencilla
int	Entero
long	Entero largo
short	Entero corto

Tabla 2.1 Tipos de datos primitivos integrados de Java

Enteros

Java define cuatro tipos de enteros: **byte**, **short**, **int** y **long**. A continuación se muestran:

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

Como se muestra en la tabla, todos los tipos de enteros tienen valores de signo positivo y negativo. Java no soporta enteros sin signo (sólo positivos). Muchos otros lenguajes de cómputo soportan enteros con signo y sin signo. Sin embargo, los diseñadores de Java sintieron que los enteros sin signo eran innecesarios.

Técnicamente, el sistema en tiempo de ejecución de Java puede usar cualquier tamaño que quiera para almacenar un tipo primitivo. Sin embargo, en todos los casos, los tipos deben actuar como está especificado.

El tipo de entero más usado es **int**. Las variables de tipo **int** suelen emplearse para bucles de control, para indicar matrices y para realizar operaciones matemáticas de propósito general.

Cuando necesite un entero que tenga un rango mayor que **int**, use **long**. Por ejemplo, he aquí un programa que calcula el número de pulgadas cúbicas que contiene un cubo de una milla por lado.

```
/*
   Calcula el número de pulgadas cúbicas
   en una milla cúbica.
*/
class Pulgadas {
  public static void main(String args[]) {
    long pc;
    long pm;

    pm = 5280 * 12;

    pc = pm * pm * pm;

    System.out.println("Hay " + pc +
                       " pulgadas cúbicas en una milla cúbica.");

  }
}
```

He aquí la salida del programa:

```
Hay 254358061056000 pulgadas cúbicas en una milla cúbica.
```

Evidentemente, no hubiera sido posible conservar el resultado en una variable **int**.

El tipo de entero más pequeño es el **byte**. Las variables del tipo **byte** resultan especialmente útiles cuando se trabaja con datos binarios que tal vez no sean compatibles directamente con otros tipos integrados de Java.

El tipo **short** crea un entero corto que tiene primero su byte de orden mayor (al que se le llama *big-endian*).

Tipos de punto flotante

Como se explicó en el módulo 1, los tipos de punto flotante pueden representar números que tienen componentes fraccionarios. Hay dos tipos de punto flotante, **float** y **double**, que representan números de precisión sencilla y doble, respectivamente. El tipo **float** es de 32 bits y el tipo **double** es de 64 bits de ancho.

Pregunte al experto

P: **¿Qué es *endianness*?**

R: *Endianness* describe la manera en que un entero se almacena en la memoria. Hay dos maneras posibles de almacenar. La primera almacena el byte más significativo en primer lugar. A esto se le llama big-endian. La otra almacena primero el byte menos significativo. A esto se le conoce como little-endian. Éste último es el método más común porque se usa en el procesador Pentium de Intel.

De los dos, **double** es el más usado porque todas las funciones matemáticas de la biblioteca de clases de Java usan valores **double**. Por ejemplo, el método **sqrt()**, (que se define con la clase **Math** estándar), devuelve un valor **double** que es la raíz cuadrada de su argumento **double**. Aquí, **sqrt()** se usa para calcular la longitud de la hipotenusa, dadas las longitudes de los dos lados opuestos:

```
/*
   Use el teorema de Pitágoras para
   encontrar la longitud de la hipotenusa
   dadas las longitudes de los dos lados
   opuestos.
*/
class Hipot {
  public static void main(String args[]) {
    double x, y, z;

    x = 3;
    y = 4;

    z = Math.sqrt(x*x + y*y);

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

Observe cómo se llama a **sqrt()**: va precedida por el nombre de la clase de la que es miembro.

La salida del programa se muestra aquí:

```
La hipotenusa es 5.0
```

He aquí otra explicación relacionada con el ejemplo anterior: como ya se mencionó, **sqrt()** es un miembro de la clase estándar **Math**. Observe cómo se llama a **sqrt()** (va precedida por el nombre **Math)**. Es una manera similar a cómo **System.out** precede a **println( )**. Aunque no todos los métodos estándar son nombrados especificando primero el nombre de su clase, varios de ellos sí son nombrados de este modo .

Caracteres

En Java, los caracteres no son cantidades de 8 bits como en casi todos los demás lenguajes de cómputo. En cambio, Java usa Unicode. *Unicode* define un conjunto de caracteres que puede representar todos los caracteres encontrados en el lenguaje humano. Por lo tanto, en Java, **char** es un tipo de 16 bytes sin signo que tiene un rango de 0 a 65,536. El conjunto de caracteres ASCII estándar de 8 bites es un subconjunto de Unicode y va de 0 a 127. Por consiguiente, los caracteres ASCII aún son caracteres válidos de Java.

Es posible asignar un valor a una variable de carácter al encerrar éste entre comillas. Por ejemplo, para asignar a la variable **carácter** la letra X:

```
char ch;
ch = 'X';
```

Puede dar salida a un valor **char** empleando la instrucción **println( )**. Por ejemplo, esta línea da salida al valor de **ch**:

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

Debido a que **char** es un tipo de 16 bits sin signo, es posible realizar varias manipulaciones aritméticas en una variable **char**. Por ejemplo, considere el siguiente programa:

```
// Las variables de carácter se manejan como enteros.
class CarAritDemo {
  public static void main(String args[]) {
    char ch;

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

    ch++; // incrementa ch
    System.out.println("ch es ahora " + ch);

    ch = 90; // da a ch el valor Z
    System.out.println("ch es ahora " + ch);
  }
}
```

Es posible incrementar **char**

A **char** puede asignársele un valor entero.

Pregunte al experto

P: **¿Por qué Java usa Unicode?**

R: Java se diseñó para usarse en todo el mundo. Por lo tanto, necesita utilizar un conjunto de caracteres que pueda representar todos los lenguajes del mundo. Unicode es el conjunto de caracteres estándar diseñado expresamente para este fin. Por supuesto, el uso de Unicode resulta ineficiente para idiomas como el inglés, el alemán, el español o el francés, cuyos caracteres pueden contenerse en 8 bits. Sin embargo, es el precio que debe pagarse por la portabilidad global.

Aquí se muestra la salida generada por este programa:

```
ch contiene X
ch es ahora Y
ch es ahora Z
```

En el programa, a **ch** se la da primero el valor X. Luego se aumenta **ch**. El resultado es que ahora contiene Y, el siguiente carácter en la secuencia ASCII (y Unicode). Aunque **char** no es un tipo entero, en algunos casos puede manejarse como si lo fuera. A continuación, se le asigna a **ch** el valor 90, que es el valor de ASCII (y Unicode) que corresponde a la letra Z. Debido a que el conjunto de caracteres de ASCII ocupa los primeros 127 valores en el conjunto de caracteres de Unicode, todos los "viejos trucos" que ha usado con caracteres en el pasado funcionarán también en Java.

El tipo boolean

El tipo **boolean** representa valores de verdadero/falso. Java define los valores verdadero y falso empleando las palabras reservadas **true** y **false**. Por lo tanto, una variable o expresión de tipo **boolean** será uno de estos dos valores.

He aquí un programa que demuestra el tipo **boolean**:

```
// Demuestra valores boolean.
class BoolDemo {
  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 boolean puede controlar la instrucción if
    if(b) System.out.println("Esto se ejecuta.");

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

    // la salida de un operador relacional es un valor boolean
    System.out.println("10 > 9 es " + (10 > 9));
  }
}
```

La salida generada por este programa se muestra aquí:

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

Hay que observar aquí tres aspectos interesantes acerca de este programa. En primer lugar, como puede ver, cuando se da salida a un valor **boolean** con **println()** se despliega "true" o "false". En segundo lugar, el valor de una variable **boolean** es suficiente, en sí misma, para controlar la instrucción **if**. No es necesario escribir una instrucción **if** como ésta:

```
if(b == true) ...
```

En tercer lugar, la salida de un operador relacional, como **<**, es un valor **boolean**. Por eso la expresión **10 > 9** despliega el valor "true". Más aún, el conjunto extra de paréntesis alrededor de **10 > 9** es necesario porque el operador + tiene una mayor precedencia que >.

Comprobación de avance

1. ¿Cuáles son los tipos enteros de Java?
2. ¿Qué es Unicode?
3. ¿Qué valores puede tener una variable **boolean**?

1. Los tipos de enteros de Java son **byte**, **short**, **int** y **long**.
2. Unicode es un conjunto internacional de caracteres.
3. Las variables del tipo **boolean** son **true** o **false**.

Proyecto 2.1 ¿A qué distancia está un trueno?

Sonido.java

En este proyecto creará un programa que calcule la distancia en metros, entre un escucha y un trueno. El sonido viaja aproximadamente a 340 metros por segundo en el aire. Por lo tanto, conociendo el intervalo entre el momento en que ve el relámpago y el momento en que el sonido lo alcanza a usted podrá calcular la distancia al trueno. Para este proyecto, suponga que el intervalo es de 7.2 segundos.

Paso a paso

1. Cree un nuevo archivo llamado **Sonido.java**.
2. Para calcular la distancia, necesitará usar valores de punto flotante. ¿Por qué? Porque el intervalo, 7.2, tiene un componente fraccionario. Aunque sería posible usar un valor de tipo **float**, usaremos **double** en este ejemplo.
3. Para calcular la distancia, multiplicará 7.2 por 340. Luego asignará este valor a una variable.
4. Por último, desplegará el resultado.

He aquí el listado completo del programa **Sonido.java**:

```
/*
   Proyecto 2.1
   Calcula la distancia a un trueno
   cuyo sonido tarda 7.2 segundos
   en llegar a usted.
*/
class Sonido {
  public static void main(String args[]) {
    double dist;

    dist = 7.2 * 340;

    System.out.println("El trueno se encuentra a "
                       + dist + " metros de distancia.");

  }
}
```

5. Compile y ejecute el programa. Se desplegará el siguiente resultado:

```
El trueno se encuentra a 2440.0 metros de distancia
```

(continúa)

6. Desafío extra: si cronometra el eco podrá calcular la distancia que lo separa de un objeto grande, como una pared de roca. Por ejemplo, si aplaude y mide cuánto tiempo le toma escuchar el eco, entonces sabrá el tiempo total de ida y vuelta. Al dividir este valor entre dos se obtiene el tiempo que toma el sonido en ir en un sentido. Luego puede usar este valor para calcular la distancia hacia el objeto. Modifique el programa anterior para que calcule la distancia, suponiendo que el intervalo es el de un eco.

HABILIDAD FUNDAMENTAL 2.2

Literales

En Java, las *literales* son los valores fijos que están representados en forma legible para los humanos. Por ejemplo, el número 100 es una literal. A las literales también se les llama *constantes*. Por lo general, el uso de las literales es tan intuitivo que éstas se han empleado de una manera u otra en todos los programas anteriores de ejemplo. Ha llegado el momento de explicarlas formalmente.

Las literales de Java pueden ser de cualquiera de los tipos de datos primitivos. La manera en que cada literal se representa depende de su tipo. Como se explicó antes, las constantes de carácter se encierran entre comillas sencillas, por ejemplo 'a' y '%' son constantes de carácter.

Las constantes de entero se especifican como números sin componentes fraccionarios. Por ejemplo, 10 y –100 son constantes de entero. Las constantes de punto flotante requieren el uso del punto decimal seguido del componente fraccionario del número. Por ejemplo, 11.123 es una constante de punto flotante. Java también le permite usar una notación científica para números de punto flotante.

Como opción predeterminada, las literales enteras son del tipo **int**. Si quiere especificar una literal **long**, añada una I y una L. Por ejemplo, 12 es **int**, pero 12L es **long**.

Como opción predeterminada, las literales de punto flotante son de tipo **double**. Para especificar una literal **float**, añada una F o f a la constante. Por ejemplo, 10.19F es de tipo **float**.

Aunque las literales de entero crean un valor **int** como opción predeterminada, pueden asignarse todavía a variables del tipo **char**, **byte** o **short**, siempre y cuando el valor que se asigne pueda representarse con el tipo de destino. Una literal de entero siempre puede asignarse a una variable **long**.

Constantes hexadecimales y octales

Como tal vez ya lo sabe, en programación a veces resulta más fácil usar un sistema numérico basado en 8 o 16 en lugar de 10. Al sistema numérico basado en 8 se le llama *octal* y usa del dígito 0 al 7. En octal, el número 10 es el mismo que el 8 en decimal. Al sistema numérico de base 16 se le llama *hexadecimal* y usa del dígito 0 al 9 y de las letras A a la F, que representan 10, 11, 12, 13, 14 y 15. Por ejemplo, el número hexadecimal 10 es 16 en decimal. Debido a la frecuencia con que se usan estos dos sistemas numéricos, Java le permite especificar constantes de entero en hexadecimal u octal en lugar de decimal. Una constante hexadecimal debe empezar con 0x (un cero seguido por una x). Una constante octal empieza con un cero. He aquí algunos ejemplos:

```
hex = 0xFF; // 255 en decimal
oct = 011; // 9 en decimal
```

Secuencias de escape de caracteres

Encerrar constantes de carácter entre comillas sencillas funciona para con casi todos los caracteres de impresión; sin embargo, unos cuantos caracteres, como los retornos de carro, plantean un problema especial cuando se usa un editor de textos. Además, otros caracteres, como las comillas sencillas y las dobles, tienen un significado especial en Java, de modo que no puede usarlos directamente. Por estas razones, Java proporciona las *secuencias de escape* especiales (en ocasiones denominadas constantes de carácter de diagonal invertida) que se muestran en la tabla 2.2. Estas secuencias se usan en lugar de los caracteres que representan.

Por ejemplo, lo siguiente asigna a **ch** el carácter de tabulador:

```
ch = '\t';
```

El siguiente ejemplo asigna una comilla sencilla a **ch**:

```
ch = '\'';
```

Literales de cadena

Java soporta otro tipo de literal: la cadena. Una *cadena* es un conjunto de caracteres encerrados entre comillas dobles. Por ejemplo:

```
"esto es una prueba"
```

Secuencia de escape	Descripción
\'	Comilla sencilla
\"	Comillas dobles
\\	Diagonal invertida
\r	Retorno de carro
\n	Nueva línea
\f	Avanzar línea
\t	Tabulador horizontal
\b	Retroceso
ddd	Constante octal (donde *ddd* es una constante octal)
\u*xxxx*	Constante hexadecimal (donde xxxx es una constante hexadecimal)

Tabla 2.2 Secuencias de escape de carácter

es una cadena. Usted ha visto ejemplos de cadenas en muchas instrucciones **println()** en los programas anteriores de ejemplo.

Además de los caracteres normales, una literal de cadena también puede contener una o más de las secuencias de escape que se han descrito. Por ejemplo, tome en cuenta el siguiente programa: éste utiliza las secuencias de escape **\n** y **\t**.

```
// Demuestra secuencias de escape en cadenas.
class CadenaDemo {
  public static void main(String args[]) {
    System.out.println("Primera línea\nSegunda línea");
    System.out.println("A\tB\tC");
    System.out.println("D\tE\tF");
  }
}
```

Use \n para generar una línea nueva.

Use tabuladores para alinear la salida.

La salida se muestra aquí:

```
Primera línea
Segunda línea
A       B       C
D       E       F
```

Observe que la secuencia de escape **\n** se usa para generar una nueva línea. No necesita usar varias instrucciones **println()** para obtener una salida de varias líneas. Sólo inserte **\n** dentro de una cadena más larga en los puntos donde quiera que se inserten las nuevas líneas.

Comprobación de avance

1. ¿Cuál es el tipo de la literal 10? ¿Cuál es el tipo de la literal 10.0?
2. ¿Cómo especifica una literal **long**?
3. ¿La "x" es una literal de cadena o de carácter?

1. La literal 10 es **int** y 10.0 es **double**.
2. Una literal **long** se especifica al agregar el sufijo L o I. Por ejemplo, 100L.
3. La literal "x" es una cadena.

Pregunte al experto

P: **¿Una cadena de un solo carácter es lo mismo que una literal de carácter? Por ejemplo, ¿"k" es lo mismo que 'k'?**

R: No. No debe confundir cadenas con caracteres. Una literal de carácter representa una sola letra de tipo **char**. Una cadena que sólo contiene una letra es todavía una cadena. Aunque las cadenas están formadas por caracteres, no son del mismo tipo.

HABILIDAD FUNDAMENTAL 2.3

Una revisión detallada de las variables

Las variables se presentaron en el módulo 1. Aquí las revisaremos de manera detallada. Como se indicó, las variables se declaran usando esta forma de instrucción:

tipo nombre-var;

donde *tipo* es el tipo de datos de la variable y *nombre-var* es el nombre de la variable. Puede declarar una variable de cualquier tipo válido, incluidos los tipos simples que se acaban de describir. Cuando crea una variable, está creando una instancia de su tipo. Por lo tanto, las capacidades de una variable están determinadas por su tipo. Por ejemplo, una variable de tipo **boolean** no puede utilizarse para almacenar valores de punto flotante. Más aún, el tipo de una variable no puede cambiar durante su existencia. Una variable **int** no puede convertirse en una **char**, por ejemplo.

Todas las variables en Java deben declararse antes de ser utilizadas. Esto es necesario porque el compilador debe saber qué tipo de datos contiene una variable antes de poder compilar apropiadamente cualquier instrucción que emplee la variable. También le permite a Java realizar una revisión estricta del tipo.

Inicialización de una variable

En general, debe proporcionar a una variable un valor antes de usarla. Una manera de hacerlo es mediante una instrucción de asignación, como ya lo ha visto. Otra manera consiste en proporcionarle un valor inicial cuando se declara. Para ello, coloque un signo de igual después del nombre de la variable y luego incluya el valor asignado. Aquí se muestra la forma general de inicialización:

tipo var = *valor*;

En este caso, *valor* es el valor que se le da a *var* cuando se crea. El valor debe ser compatible con el tipo especificado. He aquí algunos ejemplos:

```
int cuenta = 10; // proporciona  a cuenta un valor inicial de 10
char ch = 'X';   // inicializa ch con la letra X
float f = 1.2F;  // f se inicializa con 1.2
```

Cuando se declaran dos o más variables del mismo tipo empleando una lista separada por comas, puede dar a una o más de estas variables un valor inicial. Por ejemplo:

```
int a, b = 8, c = 19, d; // b y c tienen inicializaciones
```

En este caso, sólo **b** y **c** están inicializadas.

Inicialización dinámica

Aunque en los ejemplos anteriores sólo se han usado constantes como inicializadores, Java permite que las variables se inicialicen dinámicamente empleando cualquier expresión válida en el momento en que se declara la variable. Por ejemplo, he aquí un programa corto que calcula el volumen de un cilindro dados el radio de su base y su altura.

```
// Demuestra la inicialización dinámica.
class InicDin {
    public static void main(String args[]) {
      double radio = 4, altura = 5;

      // inicializa dinámicamente el volumen
      double volumen = 3.1416 * radio * radio * altura;

      System.out.println("El volumen es " + volumen);
    }
}
```

el **volumen** se inicializa dinámicamente en el tiempo de ejecución.

En este caso, están declaradas tres variables locales (**radio**, **altura** y **volumen**). Las primeras dos (**radio** y **altura**) están inicializadas. Sin embargo, **volumen** está inicializada dinámicamente al volumen del cilindro. La clave aquí es que la expresión de inicialización puede usar cualquier elemento válido en el tiempo de la inicialización, incluidas llamadas a métodos, así como otras variables o literales.

HABILIDAD FUNDAMENTAL 2.4

El alcance y la vida de las variables

Hasta ahora, todas las variables que hemos usado se declararon al principio del método **main()**. Sin embargo, Java permite que las variables se declaren dentro de cualquier bloque. Como se explicó en el módulo 1, un bloque empieza con una llave de apertura y termina con una llave de cierre. Un bloque define un *alcance*; por lo tanto, cada vez que inicia un nuevo bloque, está creando un nuevo alcance. Un alcance determina cuáles objetos son visibles a otras partes de su programa. Asimismo, determina la vida de dichos objetos.

Casi todos los demás lenguajes de cómputo definen dos categorías generales de alcance: global y local. Aunque estén soportadas por Java, no constituyen las mejores formas de categorizar los alcances de Java. Los alcances más importantes son los definidos por una clase y los definidos por un método. Más adelante en este libro, cuando se describan las clases, se incluye un análisis sobre el alcance de clase (y las variables declaradas en él). Por el momento sólo examinaremos los alcances definidos por un método o dentro de él.

El alcance definido por un método inicia con su llave de apertura. Sin embargo, si dicho método contiene parámetros, éstos se incluyen también dentro de su alcance.

Como regla general, las variables que están declaradas dentro de un alcance no son visibles (es decir, accesibles) a un código que esté definido fuera de este alcance. Por lo tanto, cuando declara una variable dentro de un alcance, localiza esa variable y la protege de acceso no autorizado, de modificación o de ambas posibilidades. Por supuesto, las reglas de alcance proporcionan la base del encapsulamiento.

Asimismo, los alcances pueden estar anidados. Por ejemplo, cada vez que crea un bloque de código, crea un nuevo alcance anidado. Cuando esto ocurre, el alcance exterior encierra al interior. Esto significa que los objetos declarados en el alcance exterior se volverán visibles al código dentro del alcance interno. Sin embargo, la situación contraria no aplica: los objetos declarados dentro del alcance interior no serán visibles fuera de él.

Para comprender el efecto de los alcances anidados, considere el siguiente programa:

```
// Demuestra el alcance del bloque.
class AlcanceDemo {
  public static void main(String args[]) {
    int x; // conocido a todo el código en main

    x = 10;
    if(x == 10) { // inicia nuevo alcance

      int y = 20; // conocido sólo a este bloque

       // x y y son conocidos aquí.
```

```
      System.out.println("x y y: " + x + " " + y);
      x = y * 2;
    }
    // y = 100; // Error. A y no se le conoce aquí ← Aquí y está fuera de su alcance.

    // x es aún conocido aquí.
    System.out.println("x es " + x);
  }
}
```

Como lo indica el comentario, la variable **x** se declara al principio del alcance de **main()** y es accesible a todo el código posterior dentro de **main()**. Dentro del bloque **if**, está declarada **y**. Debido a que un bloque define un alcance, **y** sólo es visible dentro del código de su bloque. Por tal motivo, fuera de su bloque, la línea **y = 100**; además, está convertida en comentario para que no se ejecute. Si elimina el símbolo de inicio de comentario, ocurrirá un error en tiempo de ejecución porque **y** no es visible fuera de su bloque. Dentro del bloque **if**, **x** puede usarse porque el código dentro del bloque (es decir, un alcance anidado) tiene acceso a variables declaradas por un alcance incluido.

Dentro de un bloque, las variables pueden declararse en cualquier punto; sin embargo, sólo son válidas después de ser declaradas. De manera que, si define una variable al inicio de un método, ésta estará disponible para todo el código dentro del método. Por el contrario, si declara una variable al final de un bloque, ésta carecerá de uso porque ningún código tendrá acceso a ella.

A este respecto, hay que recordar otro elemento: las variables se crean cuando se introduce su alcance y se destruyen cuando se abandona su alcance. Esto significa que una variable no contendrá su valor una vez que se haya salido de su alcance. Por lo tanto, las variables declaradas dentro de un bloque perderán su valor cuando se abandone el bloque. Así, la vida de una variable está confinada a su alcance.

Si la declaración de una variable incluye un inicializador, esa variable se reinicializará cada vez que se ingrese en el bloque en el que está declarada. Por ejemplo, considere este programa:

```
// Demuestra la vida de una variable.
class VarInicDemo {
  public static void main(String args[]) {
    int x;

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

Aquí se muestra la salida generada por este programa:

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

Como verá, **y** siempre se reinicializa a –1 cada vez que se entra en el bucle **for** interno. Aunque después se le asigna el valor 100, este valor se pierde.

Las reglas de alcance de Java tienen una peculiaridad que podría resultarle sorprendente: aunque los bloques pueden estar anidados, ninguna variable declarada dentro de un alcance interno puede tener el mismo nombre que una variable declarada por un alcance incluido. Por ejemplo, el siguiente programa, el cual trata de declarar dos variables separadas con el mismo nombre, no se compilará.

```
/*
   Este programa trata de declarar una variable
   en un alcance interno con el mismo nombre de
   una definida en un alcance externo.

   *** Este programa no se compilará. ***
*/
class VarAnid {
  public static void main(String args[]) {
    int cuenta;

    for(cuenta = 0; cuenta < 10; cuenta = cuenta+1) {
      System.out.println("Esta es la cuenta: " + cuenta);

      int cuenta; // Es ilegal.
      for(cuenta = 0; cuenta < 2; cuenta++)
        System.out.println("¡Este programa tiene un error!");
    }
  }
}
```

No es posible declarar otra vez **cuenta** porque ya está declarada.

Si está familiarizado con C/C++, sabrá entonces que no existen restricciones en cuanto a los nombres que puede asignar a las variables declaradas en un alcance interno. Por consiguiente, en C/C++ la declaración de **cuenta** dentro del bloque del bucle externo **for** es completamente válida. De hecho, una declaración de este tipo oculta la variable externa. Los diseñadores de Java sintieron que este *ocultamiento del nombre* podría conducir fácilmente a errores de programación, así que lo inhabilitaron.

Comprobación de avance

1. ¿Qué es el alcance? ¿Cómo puede crearse?
2. ¿En qué lugar de un bloque puede declararse una variable?
3. En un bloque, ¿cuándo se crea una variable?, ¿cuándo se destruye?

Operadores

Java proporciona un entorno rico en operadores. Un *operador* es un símbolo que le indica al compilador que realice una manipulación matemática o lógica específica. Java cuenta con cuatro clases generales de operadores: aritméticos, de bitwise, relacionales y lógicos. Java también define algunos operadores adicionales que manejan ciertas situaciones especiales. En este módulo se examinarán los operadores aritméticos, relacionales y lógicos. De igual forma, examinaremos también la asignación de operadores. Los operadores de bitwise, así como otros especiales, se examinarán más adelante.

HABILIDAD FUNDAMENTAL 2.5

Operadores aritméticos

Java define los siguientes operadores aritméticos:

Operador	Significado
+	Suma
–	Resta (también menos unario)
*	Multiplicación
/	División
%	Módulo
++	Incremento
—	Decremento

1. El alcance define la visibilidad y la vida de un objeto. Un bloque define un alcance.
2. Una variable puede definirse en cualquier punto dentro de un bloque.
3. Dentro de un bloque, una variable se crea cuando se encuentra su declaración y se destruye cuando se abandona el bloque.

Los operadores +, –, * y / funcionan de la misma manera en Java que en cualquier otro lenguaje de cómputo (o en el álgebra, si es el caso). Estos operadores pueden aplicarse a cualquier tipo numérico de datos y usarse en objetos de tipo **char**.

Aunque las acciones de los operadores aritméticos le resultan familiares a todos los lectores, existen unas cuantas situaciones especiales que requieren cierta explicación. En primer lugar, recuerde que cuando / se aplica a un entero, cualquier resto se truncará: por ejemplo, 10/3 será igual a 3 en una división entre enteros. Para obtener la fracción de la división debe usar el operador de módulo **%**. Este operador funciona de la misma forma en Java que en otros lenguajes: presenta el sobrante de una división entre enteros como, por ejemplo, 10 % 3 es 1. En Java, el **%** puede aplicarse a tipos enteros y de punto flotante. Por lo tanto, 10.0 % 3.0 también es 1. El siguiente programa demuestra el operador de módulo.

```
// Demuestra el operador %.
class ModDemo {
  public static void main(String args[]) {
    int iresult, irest;
    double dresult, drest;


    iresult = 10 / 3;
    irest = 10 % 3;


    dresult = 10.0 / 3.0;
    drest = 10.0 % 3.0;


    System.out.println("Resultado y sobrante de 10 / 3: " +
                       iresult + " " + irest);
    System.out.println("Resultado y sobrante de 10.0 / 3.0: " +
                       dresult + " " + drest);


  }
}
```

A continuación se muestra la salida del programa:

```
Resultado y sobrante de 10 / 3: 3 1
Resultado y sobrante de 10.0 / 3.0: 3.3333333333333335 1.0
```

Como verá, **%** presenta un sobrante de 1 para ambas operaciones, de entero o de punto flotante.

Incremento y decremento

Introducidos en el módulo 1, ++ y −− son los operadores de incremento y decremento de Java. Como verá, tienen algunas propiedades especiales que los hacen muy interesantes. Empecemos por revisar de manera precisa las acciones que los operadores de incremento y decremento llevan a cabo.

El operador de incremento agrega 1 a su operando y el de decremento resta 1. De ahí que,

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

es lo mismo que

```
x++;
```

y

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

es lo mismo que

```
--x;
```

Ambos operadores pueden preceder (prefijo) o seguir (sufijo) al operando. Por ejemplo,

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

puede escribirse como

```
++x; // forma de prefijo
```

o como

```
x++; // forma de sufijo
```

En el siguiente ejemplo, el que el incremento se aplique como prefijo o como sufijo no representa ninguna diferencia. Sin embargo, se presenta una diferencia importante cuando se usa un incremento o decremento como parte de una expresión más larga. Cuando un operador de incremento o decremento precede a su operando, Java realiza la operación correspondiente antes de obtener el valor del operando con el fin de que el resto de la expresión use dicho valor. Si el operador sigue a su operando, Java obtendrá el valor del operando antes de incrementarlo o decrementarlo. Tome en cuenta lo siguiente:

```
x = 10;
y = ++x;
```

En este caso, **y** será 11. Sin embargo, si el código se escribe como

```
x = 10;
y = x++;
```

entonces **y** será 10. En ambos casos, **x** aún tendrá un valor de 11; la diferencia es el momento en que ocurre. Tener la capacidad de controlar el momento en que la operación de incremento o decremento tiene lugar implica ventajas importantes.

HABILIDAD FUNDAMENTAL 2.6

Operadores relacionales y lógicos

En los términos *operador relacional* y *operador lógico*, el término *relacional* se refiere a las relaciones que pueden tener los valores entre sí, y *lógico* alude a la manera en que los valores verdadero y falso se conectan entre sí. Debido a que los operadores relacionales producen resultados verdaderos o falsos, a menudo trabajan con los operadores lógicos. Por tal motivo, se analizarán de manera conjunta.

A continuación se muestran los operadores relacionales:

Operador	Significado
==	Igual a
!=	No igual a
>	Mayor que
<	Menor que
>=	Mayor que o igual a
<=	Menor que o igual a

Los operadores lógicos se muestran a continuación:

Operador	Significado
&	Y
|	O
^	XO (O excluyente)
||	O de cortocircuito
&&	Y de cortocircuito
!	NO

La salida de los operadores relacionales y lógicos es un valor **boolean**.

En Java es posible comparar todos los objetos para saber si son iguales o no empleando == y **!=**. Sin embargo, los operadores de comparación <, >, <= o >= sólo pueden aplicarse a los tipos que soportan una relación de orden. Por lo tanto, todos los operadores relacionales pueden aplicarse a todos los tipos numéricos y a tipos **char**. Sin embargo, los valores de tipo **boolean** sólo pueden compararse para igualdad o desigualdad porque los valores **true** y **false** no están ordenados. Por ejemplo, **true > false** no tiene significado en Java.

Para los operadores lógicos, los operandos deben ser de tipo **boolean** y el resultado de una operación lógica es de tipo **boolean**. Los operadores lógicos, **&**, |, ^ y **!** soportan las operaciones lógicas básicas Y, O, XO y NO, de acuerdo con la siguiente tabla de verdad.

p	q	p & q	p | q	p ^ q	!p
Falso	Falso	Falso	Falso	Falso	Verdadero
Verdadero	Falso	Falso	Verdadero	Verdadero	Falso
Falso	Verdadero	Falso	Verdadero	Verdadero	Verdadero
Verdadero	Verdadero	Verdadero	Verdadero	Falso	Falso

Como se muestra en la tabla, la salida de una operación O excluyente es verdadera cuando exactamente un operador y sólo uno es verdadero.

He aquí un programa que demuestra varios de los operadores relacionales y lógicos:

```
// Demuestra los operadores relacionales y lógicos.
class OpsRelLog {
  public static void main(String args[]) {
    int i, j;
    boolean b1, b2;

    i = 10;
    j = 11;
    if(i < j) System.out.println("i < j");
    if(i <= j) System.out.println("i <= j");
    if(i != j) System.out.println("i != j");
    if(i == j) System.out.println("esto no se ejecuta");
    if(i >= j) System.out.println("esto no se ejecuta");
    if(i > j) System.out.println("esto no se ejecuta");

    b1 = true;
    b2 = false;
    if(b1 & b2) System.out.println("esto no se ejecuta");
    if(!(b1 & b2)) System.out.println("!(b1 & b2) es verdadero");
    if(b1 | b2) System.out.println("b1 | b2 es verdadero");
    if(b1 ^ b2) System.out.println("b1 ^ b2 es verdadero");
  }
}
```

Ésta es la salida del programa:

```
i < j
i <= j
i != j
!(b1 & b2) es verdadero
b1 | b2 es verdadero
b1 ^ b2 es verdadero
```

Operadores lógicos de cortocircuito

Java proporciona versiones especiales de *cortocircuito* de sus operadores lógicos Y y O que pueden usarse para producir un código más eficiente. Para comprender el porqué, considere lo siguiente. En una operación Y, si el primer operando es falso, la salida es falsa sin importar cuál sea el valor del segundo operando. En una operación O, si el primer operando es verdadero, el resultado de la operación es verdadero sin importar cuál sea el valor del segundo operando. Por lo tanto, en estos dos casos no hay necesidad de evaluar el segundo operando. Al no evaluar el segundo operando, se ahorra tiempo y se produce un código más eficiente.

El operador Y de cortocircuito es **&&** y el operador O de cortocircuito es ||; sus contrapartes normales son **&** y |. La única diferencia entre las versiones normales y las de cortocircuito es que los operandos normales siempre evaluarán cada operando, pero las versiones de cortocircuito sólo evaluarán el segundo operando cuando sea necesario.

He aquí un programa que demuestra el operador Y de cortocircuito. El programa determina si el valor de **d** es un factor de **n**. Lo hace al realizar una operación de módulo. Si el sobrante de **n / d** es cero, entonces **d** es un factor. Sin embargo, como la operación de módulo incluye una división, la forma de cortocircuito de Y se usa para evitar un error de división entre cero.

```
// Demuestra los operadores de cortocircuito.
class CCOps {
  public static void main(String args[]) {
    int n, d, q;

    n = 10;
    d = 2;
    if(d != 0 && (n % d) == 0)
      System.out.println(d + " es un factor de " + n);

    d = 0; // ahora, fije d en cero

    // Como d es cero, el segundo operando no se evalúa.
    if(d != 0 && (n % d) == 0)
      System.out.println(d + " es un factor de " + n);

    /* Ahora, intente lo mismo sin el operador de cortocircuito.
```

El operador de cortocircuito evita una división entre cero.

```
      Esto causará un error de división entre cero.
    */
    if(d != 0 & (n % d) == 0)
      System.out.println(d + " es un factor de " + n);
  }
}
```

Ahora se evalúan ambas expresiones, lo que permite que ocurra una división entre cero.

Para evitar una división entre cero, la instrucción **if** revisa primero si **d** es igual a cero. Si lo es, el Y de cortocircuito se detiene en ese punto y no realiza la división de módulo. Por lo tanto, en la primera prueba **d** es 2 y se realiza la operación de módulo. La segunda prueba falla porque **d** se fija en cero y la operación de módulo se detiene, lo que evita un error de división entre cero. Por último, se prueba el operador normal Y. Esto ocasiona que ambos operandos se evalúen, lo que conduce a un error en tiempo de ejecución cuando la división entre cero ocurre.

Comprobación de avance

1. ¿Qué hace el operador **%**? ¿A qué tipos puede aplicarse?
2. ¿Qué tipo de valores pueden emplearse como operandos de los operadores lógicos?
3. ¿Un operador de cortocircuito siempre evalúa sus dos operandos?

HABILIDAD FUNDAMENTAL 2.7

El operador de asignación

Ha estado utilizando el operador de asignación desde el módulo 1. Es tiempo de que lo analice de manera más formal. El *operador de asignación* es el signo de igual, =. Este operador funciona en Java de manera muy parecida a cualquier otro lenguaje de cómputo. Tiene esta forma general:

var = expresión.

Aquí, el tipo de *var* debe ser compatible con el tipo de *expresión*.

1. El **%** es el operador de módulo, el cual devuelve el sobrante de una división entre enteros. Puede aplicarse a todos los tipos numéricos.
2. Los operadores lógicos deben tener operandos de tipo **boolean**.
3. No, un operador de cortocircuito evalúa su segundo operando sólo si el resultado de la operación no puede determinarse únicamente mediante su primer operando.

Pregunte al experto

P: **Debido a que los operadores de cortocircuito son, en algunos casos, más eficientes que sus contrapartes normales, ¿por qué Java continúa ofreciendo los operadores Y y O normales?**

R: En algunos casos es posible que desee que ambos operandos de una operación Y u O se evalúen debido a los efectos colaterales producidos. Considere lo siguiente:

```
// Los efectos colaterales pueden ser importantes.
class EfectosColate {
  public static void main(String args[]) {
    int i;

    i = 0;

     /* Aquí, i todavía se incrementa aunque
        falle la instrucción if. */
    if(false & (++i < 100))
       System.out.println("esto no se despliega");
    System.out.println("Las instrucciones if se ejecutan: " + i); //
despliega 1

    /* En este caso, i no se incrementa porque
       el operador de cortocircuito salta el incremento. */
    if(false && (++i < 100))
      System.out.println("esto no se despliega");
    System.out.println("Las instrucciones if se ejecutan: " + i); // aún es 1!!
  }
}
```

Como el comentario lo indica, en la primera instrucción **if**, **i** se incrementa si el **if** tiene éxito o no. Sin embargo, cuando se usa el operador de cortocircuito, la variable **i** no se incrementa cuando el primer operando es falso. En este caso la lección es que si su código espera el operando de la derecha de una operación Y u O, debe usar las formas de estas operaciones que no sean de cortocircuito.

El operador de asignación tiene un atributo interesante con el que tal vez no esté familiarizado: le permite crear una cadena de asignaciones. Por ejemplo, considere este fragmento:

```
int x, y, z;

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

Este fragmento asigna el valor 100 a las variables **x**, **y** y **z** empleando una sola instrucción, lo que funciona debido a que = es un operador que arroja el valor de la expresión de la derecha. Por

consiguiente, el valor de **z = 100** es 100, que luego se asigna a **y**, que a su vez se asigna a **z**. El empleo de una "cadena de asignación" constituye una manera fácil de asignar un valor común a un grupo de variables.

HABILIDAD FUNDAMENTAL 2.8

Asignaciones de método abreviado

Java proporciona operadores especiales de asignación de *método abreviado* que simplifican la codificación de ciertas instrucciones de asignación. Empecemos con un ejemplo: la instrucción de asignación que se muestra aquí

```
x = x + 10;
```

puede escribirse empleando el método abreviado de Java, como

```
x += 10;
```

El par de operadores += le indican al compilador que asigne a **x** el valor de **x** más 10.

He aquí otro ejemplo. La instrucción

```
x = x - 100;
```

es lo mismo que

```
x -= 100;
```

Ambas instrucciones asignan a **x** el valor de **x** menos 100.

Este método abreviado funcionará para todos los operadores binarios en Java (es decir, los que requieren dos operandos). La forma general del método abreviado es

var op = expresión

Por lo tanto, los operadores de asignación aritméticos y lógicos son los siguientes:

+=	**−=**	***=**	**/=**
%=	**&=**	**|=**	**^=**

Debido a que estos operadores combinan una operación con una asignación, se les denominaba *operadores de asignación compuestos*.

Los operadores de asignación compuestos proporcionan dos beneficios. En primer lugar, son más compactos que sus equivalentes "largos"; en segundo lugar, el sistema en tiempo de ejecución de Java los implementa de manera más eficiente. Por estas razones, a menudo notará que los programas de Java profesionalmente escritos emplean operadores de asignación compuestos.

HABILIDAD FUNDAMENTAL 2.9

Conversión de tipos en asignaciones

En programación, es común asignar un tipo de variable a otro; por ejemplo, tal vez quiera asignar un valor **int** a una variable **float**, como se muestra aquí:

```
int i;
float f;

i = 10;
f = i; // asigna una int a una float
```

Cuando se combinan tipos compatibles en una asignación, el valor de la derecha se convierte automáticamente al tipo de la izquierda. Por lo tanto, en el fragmento anterior, el valor de **i** se convierte en **float** y luego se asigna a **f**. Sin embargo, debido a la revisión estricta que Java realiza de los tipos, no todos ellos son compatibles y, por consiguiente, no todas las conversiones de tipo se permiten implícitamente. Por ejemplo, **boolean** e **int** no son compatibles.

Cuando un tipo de datos se asigna a otro tipo de variable, una *conversión automática de tipo* tendrá lugar si

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

Cuando estas dos condiciones se cumplen, tiene lugar una *conversión de ensanchamiento*. Por ejemplo, el tipo **int** siempre es lo suficientemente grande como para contener todos los valores válidos de **byte**; y tanto **int** como **byte** son tipos enteros, de modo que puede aplicarse una conversión automática de **byte** a **int**.

En el caso de las conversiones de ensanchamiento, los tipos numéricos, incluidos los tipos de entero y de punto flotante, son compatibles entre sí. Por ejemplo, el siguiente programa es perfectamente válido porque la conversión de ensanchamiento de **long** a **double** se realiza automáticamente.

```
// Demuestra la conversión automática de long a double.
class LAD {
  public static void main(String args[]) {
    long L;
    double D;

    L = 100123285L;
    D = L; <-------------------------- Conversión automática de long a double

    System.out.println("L y D: " + L + " " + D);

  }
}
```

Aunque hay una conversión automática de **long** a **double**, no hay una conversión automática de **double** a **long** porque no se presenta una conversión de ensanchamiento. Por lo tanto, la siguiente versión del programa anterior no es válida.

```
// *** Este programa no se compilará. ***
class LAD {
  public static void main(String args[]) {
    long L;
    double D;

    D = 100123285.0;
    L = D; // ¡¡¡Ilegal!!! ← No hay conversión automática de double a long.

    System.out.println("L y D: " + L + " " + D);

  }
}
```

No hay conversiones automáticas de tipos numéricos a **char** o **boolean**. Además, **char** y **boolean** no son compatibles entre sí. Sin embargo, es posible asignar una literal de entero a **char**.

HABILIDAD FUNDAMENTAL 2.10

Moldeado de tipos incompatibles

Si bien las conversiones automáticas de tipo son útiles, no satisfacen todas las necesidades de programación porque se aplican sólo a conversiones de ensanchamiento entre tipos compatibles. En todos los demás casos, debe emplearse el moldeado. El *moldeado* es una instrucción para que el compilador convierta un tipo en otro; así pues, solicita una conversión explícita de tipo. Un moldeado tiene esta forma general:

(*tipo-destino*) *expresión*

En este caso, *tipo-destino* especifica el tipo deseado al que se convertirá la expresión especificada. Por ejemplo, si quiere convertir el tipo de la expresión **x/y** a **int** puede escribir

```
double x, y;
// ...
(int) (x/y)
```

Aquí, aunque **x** y **y** son de tipo **double**, el molde convierte la salida de la expresión a **int**. Los paréntesis que rodean a **x/y** son necesarios, pues de otra manera el moldeado a **int** sólo se aplicaría a la **x** y no al resultado de la división. En este caso el moldeado es necesario porque no hay conversión automática de **double** a **int**.

Cuando un moldeado incluye una *conversión de estrechamiento*, podría perderse información. Por ejemplo, cuando se convierte **long** en **short**, se perderá información si el valor de **long** es mayor

que el rango de **short** debido a que se eliminarán los bits de orden superior. Cuando un valor de punto flotante se moldea a un tipo de entero, el componente fraccionario también se perderá debido al truncamiento. Por ejemplo, si el valor 1.23 se asigna a un entero, el valor resultante será simplemente 1. Se perderá el 0.23.

El siguiente programa demuestra algunos tipos de conversiones que requieren moldeado:

```
// Demuestra el moldeado.
class MoldeDemo {
  public static void main(String args[]) {
    double x, y;
    byte b;
    int i;
    char ch;

    x = 10.0;
    y = 3.0;
    i = (int) (x / y); // moldea double a int
    System.out.println("Salida de entero de x / y: " + i);

    i = 100;
    b = (byte) i;
    System.out.println("Valor de b: " + b);

    i = 257;
    b = (byte) i;
    System.out.println("Valor de b: " + b);

    b = 88; // código ASCII para X
    ch = (char) b;
    System.out.println("ch: " + ch);
  }
}
```

En esta conversión habrá truncamiento.

No hay pérdida de info aquí. Un **byte** puede contener el valor 100.

Esta vez se pierde información. Un **byte** no puede contener el valor 257.

Moldeado entre tipos incompatibles.

A continuación se muestra la salida del programa:

```
Salida de entero de x / y: 3
Valor de b: 100
Valor de b: 1
ch: X
```

En el programa, el moldeado de **(x/y)** a **int** da como resultado el truncamiento del componente fraccionario y la pérdida de información. A continuación, no ocurre pérdida de información cuando se asigna a **b** el valor 100, pues un **byte** puede contener el valor 100. Sin embargo, cuando se hace el intento de asignar el valor 257 a **b**, se perderán datos porque 257 excede el valor máximo de **byte**. Por último, cuando se asigna un valor **byte** a un **char** no se pierde información pero se requiere un moldeado.

Comprobación de avance

1. ¿Qué es el moldeado?
2. ¿Puede asignarse una variable **short** a una **int** sin moldeado? ¿Puede asignarse una **byte** a una **char** sin moldeado?
3. ¿Cómo reescribiría la siguiente instrucción?

```
x = x + 23;
```

Precedencia de operadores

En la siguiente tabla se muestra, del mayor al menor, el orden de precedencia para todos los operadores de Java. Esta tabla incluye varios operadores que se analizarán más adelante en este libro.

Mayor			
( )	[ ]	.	
++	--	~~	!
*	/	%	
+	-		
>>	>>>	<<	
>	>=	<	<=
==	!=		
&			
^^			
|			
&&			
||			
?:			
=	op=		
Menor			

1. Un moldeado es una conversión explícita.
2. Sí. No.
3. x += 23.

Proyecto 2.2 Despliegue una tabla de verdad para los operadores lógicos

OpLogicTabla.java

En este proyecto creará un programa que despliegue la tabla de verdad de los operadores lógicos de Java. Debe hacer que las columnas de la tabla estén alineadas. En este proyecto se utilizarán varias de las funciones vistas en este módulo, incluidas las secuencias de escape de Java y los operadores lógicos. También se ilustran las diferencias respecto a la precedencia entre el operador aritmético + y los operadores lógicos.

Paso a paso

1. Cree un nuevo archivo llamado **OpLogicTabla.java**.
2. Para asegurar que las columnas se alinien, usará la secuencia de escape **\t** para insertar tabuladores en cada cadena de salida. Por ejemplo, la instrucción **println()** despliega el encabezado para la tabla:

```
System.out.println("P\tQ\tY\tO\tXO\tNO");
```

3. Cada línea posterior de la tabla usará tabuladores para colocar la salida de cada operación bajo su encabezado apropiado.
4. He aquí el listado completo del programa **OpLogicTabla.java**. Ingréselo en este momento.

```
/*
   Proyecto 2.2/: Imprime una tabla de verdad para operadores lógicos.
class OpLogicTabla {
  public static void main(String args[]) {

    boolean p, q;

    System.out.println("P\tQ\tY\tO\tXO\tNO");

    p = true; q = true;
    System.out.print(p + "\t" + q +"\t");
    System.out.print((p&q) + "\t" + (p|q) + "\t");
    System.out.println((p^q) + "\t" + (!p));

    p = true; q = false;
    System.out.print(p + "\t" + q +"\t");
    System.out.print((p&q) + "\t" + (p|q) + "\t");
    System.out.println((p^q) + "\t" + (!p));

    p = false; q = true;
    System.out.print(p + "\t" + q +"\t");
```

(continúa)

```
    System.out.print((p&q) + "\t" + (p|q) + "\t");
    System.out.println((p^q) + "\t" + (!p));

    p = false; q = false;
    System.out.print(p + "\t" + q +"\t");
    System.out.print((p&q) + "\t" + (p|q) + "\t");
    System.out.println((p^q) + "\t" + (!p));
  }
}
```

Observe los paréntesis que encierran a las operaciones lógicas dentro de las instrucciones **println()**. Son necesarias debido a la precedencia de los operadores de Java. El operador + es mayor que los operadores lógicos.

5. Compile y ejecute el programa. Se desplegará la siguiente tabla.

```
P       Q       Y       O       XO      NO
true    true    true    true    false   false
true    false   false   true    true    false
false   true    false   true    true    true
false   false   false   false   false   true
```

6. Por su cuenta, trate de modificar el programa para que use y despliegue 1 y 0 en lugar de true y false. ¡Tal vez requiera un poco más de esfuerzo de lo que pensaría al principio!

HABILIDAD FUNDAMENTAL 2.11

Expresiones

Los operadores, las variables y las literales integran las *expresiones*. En Java una expresión es cualquier combinación válida de estas piezas. Tal vez ya conozca la forma general de una expresión a partir de sus otras experiencias en programación o del álgebra. Sin embargo, a continuación se analizarán algunos aspectos de las expresiones.

Conversión de tipos en expresiones

Dentro de una expresión, es posible combinar dos o más tipos diferentes de datos, siempre y cuando sean compatibles entre sí. Por ejemplo, puede combinar **short** y **long** dentro de una expresión porque ambos son tipos numéricos. Cuando diferentes tipos de datos dentro de una expresión se combinan, todos se convierten en el mismo tipo. Esto se logra mediante el uso de las *reglas de promoción de tipo* de Java.

En primer lugar, todos los valores **char**, **byte** y **short** se promueven o ascienden a **int**. Luego, si un operando es **long**, toda la expresión se promueve a **long**. Si un operando es **float**, toda la expresión se promueve a **float**. Si cualquiera de los operandos es **double**, el resultado es **double**.

Resulta importante comprender que la promoción de tipo sólo se aplica a los valores que son operados cuando se evalúa una expresión. Por ejemplo, si el valor de una variable **byte** se promueve a **int** dentro de una expresión, la variable, fuera de la expresión, será aún un **byte**. La promoción de tipo sólo afecta a la evaluación de una expresión.

Sin embargo, la promoción de tipo puede conducir a resultados un tanto inesperados. Por ejemplo, cuando una operación aritmética incluye dos valores **byte**, ocurre la siguiente secuencia: en primer lugar, los operandos **byte** se promueven a **int**. Luego tiene lugar la operación arrojando un resultado **int**. Por lo tanto, la salida de una operación que incluye dos valores **byte** será un valor **int**. Tal vez esto no sea lo que espere de manera intuitiva . Considere el siguiente programa:

```
// Una sorpresa de promoción!
class PromDemo {
  public static void main(String args[]) {
    byte b;
    int i;

    b = 10;
    i = b * b; // OK, no se necesita moldeado

    b = 10;
    b = (byte) (b * b); // ¡¡se necesita moldeado!!

    System.out.println("i y b: " + i + " " + b);
  }
}
```

No se necesita moldeado porque ya se elevó a **int**.

¡Aquí es necesario el moldeado para asignar un **int** a un byte!

Un poco en contra de lo que esperaría por intuición , no se necesita moldeado cuando se asigna **b * b** a **i**, porque **b** se promueve a **int** cuando se evalúa la expresión. Sin embargo, cuando se asigna **b * b** a **b**, sí se necesita moldeado (¡de regreso a **byte**!). Tenga esto en cuenta si recibe mensajes inesperados de error sobre incompatibilidad de tipo en expresiones que, de otra manera, parecerían perfectamente correctas.

Este mismo tipo de situación ocurre también cuando se realizan operaciones en valores **char**. Por ejemplo, en el siguiente fragmento, el moldeado de regreso a **char** es necesario debido a la promoción de **ch1** y **ch2** a **int** dentro de la expresión.

```
char ch1 = 'a', ch2 = 'b';

ch1 = (char) (ch1 + ch2);
```

Sin el moldeado, el resultado de sumar **ch1** y **ch2** sería **int**, el cual no puede asignarse a **char**.

Los moldeados no sólo resultan útiles cuando se convierte entre tipos en una asignación. Por ejemplo, considere el siguiente programa: usa un moldeado a **double** para obtener el componente fraccionario de una división que de otra manera sería de entero.

```
// Uso de moldeado.
class UsoMold {
  public static void main(String args[]) {
    int i;
```

```
    for(i = 0; i < 5; i++) {
      System.out.println(i + " / 3: " + i / 3);
      System.out.println(i + " / 3 con fracciones: "
                         + (double) i / 3);
      System.out.println();
    }
  }
}
```

La salida del programa se muestra a continuación:

```
0 / 3: 0
0 / 3 con fracciones: 0.0

1 / 3: 0
1 / 3 con fracciones: 0.3333333333333333

2 / 3: 0
2 / 3 con fracciones: 0.6666666666666666

3 / 3: 1
3 / 3 con fracciones: 1.0

4 / 3: 1
4 / 3 con fracciones: 1.3333333333333333
```

Espaciado y paréntesis

Una expresión de Java puede tener tabuladores y espacios en ella para hacerla más legible. Por ejemplo, las dos siguientes expresiones son iguales, pero la segunda es más fácil de leer:

```
x=10/y*(127/x);

x = 10 / y * (127/ x);
```

Los paréntesis aumentan la precedencia de las operaciones contenidas dentro de ellos, como en el álgebra. El uso de paréntesis redundantes o adicionales no causará errores ni hará más lenta la ejecución de la expresión. Se le recomienda que use paréntesis para hacer más claro el orden exacto de la evaluación, tanto para usted como para aquellas personas que intenten saber cómo funciona su programa. Por ejemplo, ¿cuál de las dos expresiones siguientes es más fácil de leer?

```
x = y/3-34*temperatura+127;

x = (y/3) - (34*temperatura) + 127;
```

Comprobación de dominio del módulo 2

1. ¿Por qué Java especifica estrictamente el rango y el comportamiento de sus tipos primitivos?
2. ¿Cuál es el tipo de carácter de Java y en qué difiere del tipo de carácter empleado por muchos otros lenguajes de programación?
3. Un valor **boolean** puede tener cualquier valor que usted desee porque cualquier valor diferente de cero es verdadero. ¿Cierto o falso?
4. Dada esta salida,

```
Uno
Dos
Tres
```

   y empleando una sola cadena, muestre la instrucción **println()** que la produce.

5. ¿Qué está incorrecto en este fragmento?

```
for(i = 0; i < 10; i++) {
  int suma;

  suma = suma + i;
}
System.out.println("La suma es: " + suma);
```

6. Explique la diferencia entre las formas de prefijo y sufijo del operador de incremento.
7. Muestre la manera en la que un Y de cortocircuito puede usarse para evitar un error de división entre cero.
8. En una expresión, ¿cuáles tipos son promovidos a **byte** y a **short**?
9. En general, ¿cuándo es necesario el moldeado?
10. Escriba un programa que encuentre todos los números primos entre 1 y 100.
11. ¿El uso de paréntesis redundantes afecta el desempeño del programa?
12. ¿Un bloque define un alcance?

Módulo 3

Instrucciones de control del programa

HABILIDADES FUNDAMENTALES

3.1	Ingrese caracteres desde el teclado
3.2	Conozca la forma completa de la instrucción **if**
3.3	Use la instrucción **switch**
3.4	Conozca la forma completa del bucle **for**
3.5	Use el bucle **while**
3.6	Use el bucle **do-while**
3.7	Use **break** para salir de un bucle
3.8	Use **break** como una forma de goto
3.9	Aplique **continue**
3.10	Anide bucles

En este módulo aprenderá acerca de las instrucciones que controlan el flujo de ejecución de un programa. Existen dos categorías de instrucciones de control del programa: instrucciones de *selección*, que incluyen **if** y **switch**, e instrucciones de *iteración*, que incluyen los bucles **for**, **while** y **do-while**; además, hay instrucciones *de salto*, que incluyen **break**, **continue** y **return**. Excepto return, que se analizará más adelante en este libro, las restantes instrucciones de control, incluidas **if** y **for**, de las que ya se ha brindado una breve introducción, se examinarán a continuación de manera detallada. El módulo inicia con la explicación de la manera de realizar algunas entradas simples de teclado.

HABILIDAD FUNDAMENTAL 3.1

Entrada de caracteres desde el teclado

Antes de examinar las instrucciones de control de Java, haremos una pequeña digresión que le permitirá empezar a escribir programas interactivos. Hasta el momento, los programas de ejemplo de este libro han desplegado información *para* el usuario, pero no han recibido información *del* usuario. Por lo tanto, usted ha estado usando la salida de la consola, pero no la entrada (el teclado) de ésta. La principal razón para ello es que el sistema de entrada de Java depende más bien de un complejo sistema de clases, cuyo uso requiere la comprensión de varias características, como el manejo de excepciones y de clases, los cuales no se analizarán sino hasta más adelante en este libro. No existe un paralelo directo que se asemeje al muy conveniente método **println( )** que, por ejemplo, le permita leer varios tipos de datos ingresados por el usuario. Francamente, el método de Java para entradas en la consola no es muy fácil de usar. Además, la mayor parte de los programas y los applets de Java están basados en imágenes y ventanas, no en la consola. Por tal motivo, en este libro no encontrará un gran uso de la entrada de la consola. Sin embargo, hay un tipo de entrada de consola que *es* fácil de usar: la lectura de un carácter del teclado. Esta función se analizará a continuación debido a que varios de los ejemplos de este módulo hacen uso de ella. .

La manera más fácil de leer un carácter del teclado es llamar a **System.in.read( )**. **System.in** es el complemento de S**ystem.out** y es el objeto de entrada que está adjunto al teclado. El método **read( )** espera hasta que el usuario oprime una tecla y luego devuelve el resultado. El carácter es devuelto como un entero, de modo que puede moldearse como **char** para asignarlo a la variable **char**. Como opción predeterminada, la entrada de la consola se almacena en línea, de modo que debe oprimir ENTER antes de que cualquier carácter que escriba se envíe a su programa. He aquí un programa que lee un carácter del teclado.

```
// Lee un carácter del teclado.
class EnTec {
  public static void main(String args[])
    throws java.io.IOException {
```

```
    char ch;

    System.out.print("Oprima una tecla seguida de ENTER: ");

    ch = (char) System.in.read(); // obtiene un char
    System.out.println("Su tecla es: " + ch);
  }
}
```

Lee un carácter del teclado.

He aquí una ejecución de ejemplo:

```
Oprima una tecla seguida de ENTER: t
Su tecla es: t
```

En el programa, observe que **main()** empieza así:

```
public static void main(String args[])
  throws java.io.IOException {
```

Debido a que **System.in.read()** se está usando, el programa debe especificar la cláusula **throws java.in.IOException**. Esta línea es necesaria para manejar errores de entrada pues es parte del mecanismo de manejo de excepciones de Java que se analizará en el módulo 9. Por el momento, no se preocupe de su significado preciso.

El hecho de que **System.in** se almacene en cola suele representar una fuente de molestias. Cuando oprime ENTER, se introduce un retorno de carro en el flujo de entrada. Más aún, esos caracteres se quedan pendientes en la memoria temporal de entrada hasta que se leen. Por consiguiente, en el caso de algunas aplicaciones, tal vez necesite eliminarlas (al leerlas) antes de la siguiente operación de entrada. Más adelante, en este módulo, se proporcionará un ejemplo de ello.

Comprobación de avance

1. ¿Qué es **System.in**?
2. ¿Cómo puede leer un carácter escrito en el teclado?

1. **System.in** es el objeto de entrada que está vinculado con la entrada estándar, la cual suele ser el teclado.
2. Para leer un carácter, llame a **System.in.read()**.

HABILIDAD FUNDAMENTAL 3.2

La instrucción if

En el módulo 1 se introdujo la instrucción **if** y se examinó de manera detallada. La forma completa de la instrucción **if** es

if(*condición*) *instrucción*;
else *instrucción*;

donde los destinos del **if** y el **else** son instrucciones sencillas. La cláusula **else** es opcional. Los destinos de ambos pueden ser bloques de instrucciones. La forma general del **if**, empleando bloques de instrucciones, es

if(*condición*)
{
secuencia de instrucciones
}
else
{
secuencia de instrucciones
}

Si la expresión condicional es verdadera, se ejecutará el destino del **if**; de otra manera, se ejecutará, si existe, el destino del **else**. En ningún momento se ejecutarán ambas. La expresión condicional que controla el **if** debe producir un resultado **boolean**.

Para demostrar **if** (y otras tantas instrucciones de control), crearemos y desarrollaremos un simple juego computarizado de adivinanzas que sea adecuado para niños pequeños. En la primera versión del juego, el programa le pregunta al jugador una letra entre la A y la Z. Si el jugador oprime la letra correcta en el teclado, el programa responde imprimiendo el mensaje **** Correcto ****. A continuación se muestra el programa:

```
// Juego de adivinar la letra.
class Adiv {
  public static void main(String args[])
    throws java.io.IOException {

    char ch, respuesta = 'K';

    System.out.println("Estoy pensando en una letra entre la A y la Z.");
    System.out.print("Puedes adivinarla: ");

    ch = (char) System.in.read(); // lee un char del teclado

    if(ch == respuesta) System.out.println("** Correcto **");
  }
}
```

Este programa le pide una letra al jugador y luego lee un carácter del teclado. Empleando una instrucción **if**, revisa ese carácter contra la respuesta que, en este caso, es K,. Si se introdujo K, se despliega el mensaje. Cuando pruebe este programa, recuerde que la K debe ingresarse en mayúsculas.

Llevando más allá el juego de adivinanzas, la siguiente versión emplea **else** para imprimir un mensaje cada vez que se elija la letra incorrecta.

```
// Juego de adivinar la letra, 2a versión.
class Adiv2 {
  public static void main(String args[])
    throws java.io.IOException {

    char ch, respuesta = 'K';

    System.out.println("Estoy pensando en una letra entre la A y la Z.");
    System.out.print("Puedes adivinarla: ");

    ch = (char) System.in.read(); // obtiene un char

    if(ch == respuesta) System.out.println("** Correcto **");
    else System.out.println("...Lo siento, es incorrecta.");
  }
}
```

If anidados

Un ***if anidado*** es una instrucción **if** que es el destino de otro **if** o **else**. Los **if** anidados son muy comunes en programación. Lo más importante que se debe recordar acerca de los **if** en Java es que una instrucción **else** siempre alude a la instrucción **if** más cercana que se encuentre dentro del mismo bloque que el **else** y que no esté ya asociada con un **else**. He aquí un ejemplo:

```
if(i == 10) {
  if(j < 20) a = b;
  if(k > 100) c = d;
  else a = c; // este else alude a if(k > 100)
}
else a = d; // este else alude a if(1 == 10)
```

Como lo indican los comentarios, el **else** final no está asociado con **if(j < 20)** porque no se encuentra en el mismo bloque (aunque es el **if** más cercano sin un **else**). En cambio, el **else** final está asociado con **if(i == 10)**. El **else** interno alude al **if(k > 100)** porque es el **if** más cercano dentro del mismo bloque.

Puede usar un **if** anidado para agregar una mejora adicional al juego de adivinanzas. Esta adición proporciona al jugador un poco de retroalimentación acerca de suposiciones erróneas.

```
// Juego de adivinar la letra, 3a versión.
class Adiv3 {
  public static void main(String args[])
    throws java.io.IOException {

    char ch, respuesta = 'K';

    System.out.println("Estoy pensando en una letra entre la A y la Z.");
    System.out.print("Puedes adivinarla: ");

    ch = (char) System.in.read(); // obtiene un char

    if(ch == respuesta) System.out.println("** Correcto **");
    else {
      System.out.print("...Lo siento, te encuentras ");

      // un if anidado
      if(ch < respuesta) System.out.println("demasiado bajo");
      else System.out.println("demasiado alto");
    }
  }
}
```

Aquí se muestra una ejecución de ejemplo:

```
Estoy pensando en una letra entre la A y la Z.
Puedes adivinarla: Z
...Lo siento, te encuentras demasiado alto
```

La escalera if-else-if

Una construcción común de programación que se basa en el **if** anidado es la *escalera* ***if-else-if***. Tiene este aspecto:

if(*condición*)
  instrucción;
else if(*condición*)
  instrucción;
else if(*condición*)
  instrucción;

```
.
.
.
else
  instrucción;
```

Las expresiones de condición se evalúan de arriba a abajo. En cuanto se encuentra una condición verdadera, se ejecuta la instrucción asociada con ella y se omite el resto de la escalera. Si ninguna de las instrucciones es verdadera, se ejecutará la instrucción **else** final. A menudo el **else** final actúa como una condición predeterminada; es decir, si fallan todas las pruebas de condición, se ejecuta la última instrucción **else**. Si no hay un **else** final y todas las demás condiciones son falsas, no tendrá lugar acción alguna.

El siguiente programa demuestra la escalera **if-else-if**:

```
// Demuestra una escalera if-else-if.
class Escalera {
  public static void main(String args[]) {
    int x;

    for(x=0; x<6; x++) {
      if(x==1)
        System.out.println("x es uno");
      else if(x==2)
        System.out.println("x es dos");
      else if(x==3)
        System.out.println("x es tres");
      else if(x==4)
        System.out.println("x es cuatro");
      else
        System.out.println("x no se encuentra entre 1 y 4"); ← Ésta es la instrucción predeterminada.
    }
  }
}
```

El programa produce la siguiente salida:

```
x no se encuentra entre 1 y 4
x es uno
x es dos
x es tres
x es cuatro
x no se encuentra entre 1 y 4
```

Como puede ver, el **else** predeterminado se ejecuta sólo si ninguna de las instrucciones **if** anteriores tiene éxito.

Comprobación de avance

1. ¿De qué tipo debe ser la condición que controla el **if**?
2. ¿A qué **if** se asocia siempre un **else**?
3. ¿Qué es una escalera **if-else-if**?

HABILIDAD FUNDAMENTAL 3.3

La instrucción switch

La segunda de las instrucciones de selección de Java es **switch**, la cual proporciona un árbol de varias ramas; por lo tanto, permite que un programa seleccione entre varias opciones. Aunque una serie de instrucciones **if** anidadas puede realizar pruebas de varias vías, en muchas ocasiones **switch** es un método más eficiente. Funciona de la siguiente manera: el valor de una expresión se prueba sucesivamente contra una lista de constantes. Cuando se encuentra una coincidencia, se ejecuta la secuencia de instrucciones asociadas con dicha coincidencia. La forma general de la instrucción **switch** es

```
switch(expresión) {
  case constantes1;
    secuencia de instrucciones
    break;
case constantes2;
    secuencia de instrucciones
    break;
case constantes3;
    secuencia de instrucciones
    break;
.
.
.
default;
    secuencia de instrucciones
}
```

1. La condición que controla un **if** debe ser del tipo **boolean**.
2. Un **else** siempre se asocia con el **if** más cercano en el mismo bloque que no está asociado con un **else**.
3. Una escalera **if-else-if** es una secuencia de instrucciones **if-else** anidadas.

La expresión **switch** puede ser de tipo **char**, **byte**, **short** o **int**. (No se permiten, por ejemplo, expresiones de punto flotante). Con frecuencia, la expresión que controla el **switch** es simplemente una variable. Las constantes de **case** deben ser literales de un tipo compatible con la expresión. Dos constantes case en el mismo **switch** no pueden tener valores idénticos.

La secuencia de instrucciones **default** se ejecuta si ninguna constante **case** coincide con la expresión. Esta secuencia **default** es opcional; si no está presente, ninguna acción se presenta si fallan todas las coincidencias. Cuando se encuentra una coincidencia, se ejecutan las instrucciones asociadas con ese **case** hasta que se encuentra **break** o, en el caso de **default** o el último **case**, hasta que se alcance el final del **switch**.

El siguiente programa demuestra **switch**.

```
// Demuestra switch.
class SwitchDemo {
  public static void main(String args[]) {
    int i;

    for(i=0; i<10; 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;
        case 4:
          System.out.println("i es cuatro");
          break;
        default:
          System.out.println("i es cinco o mayor");
      }

  }
}
```

A continuación se muestra la salida producida por el programa:

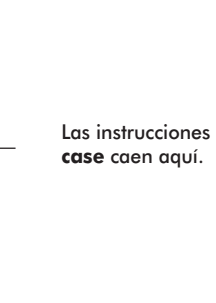
```
i es cero
i es uno
i es dos
i es tres
i es cuatro
i es cinco o mayor
i es cinco o mayor
i es cinco o mayor
i es cinco o mayor
i es cinco o mayor
```

Como puede ver, cada vez que el bucle se recorre, se ejecutan las instrucciones asociadas con la constante **case** que coinciden con **i**. Todas las demás se omiten. Cuando **i** es cinco o mayor, ninguna instrucción **case** coincide, de modo que se ejecuta la instrucción **default**.

Técnicamente, la instrucción **break** es opcional, aunque la mayor parte de las aplicaciones de **switch** la usarán. Cuando se encuentra dentro de la secuencia de la instrucción de **case**, la instrucción **break** causa que el programa fluya para salir de toda la instrucción **switch** y se reanude en la siguiente instrucción fuera de **switch**. Sin embargo, si una instrucción **break** no termina la secuencia de instrucciones asociada con **case**, entonces todas las instrucciones *que se encuentran en (y que siguen al)* **case** coincidente se ejecutarán hasta que se encuentre un **break** (o el final del **switch**).

Por ejemplo, estudie cuidadosamente el siguiente programa. Antes de observar la salida, ¿puede imaginar lo que se desplegará en la pantalla?

```
// Demuestra switch sin instrucciones break.
class NoBreak {
  public static void main(String args[]) {
    int i;

    for(i=0; i<=5; i++) {
      switch(i) {
        case 0:
          System.out.println("i es menor que uno");
        case 1:
          System.out.println("i es menor que dos");
        case 2:
          System.out.println("i es menor que tres");
        case 3:
          System.out.println("i es menor que cuatro");
        case 4:
          System.out.println("i es menor que cinco");
```



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

El programa despliega la siguiente salida:

```
i es menor que uno
i es menor que dos
i es menor que tres
i es menor que cuatro
i es menor que cinco

i es menor que dos
i es menor que tres
i es menor que cuatro
i es menor que cinco

i es menor que tres
i es menor que cuatro
i es menor que cinco

i es menor que cuatro
i es menor que cinco

i es menor que cinco
```

Como lo ilustra este programa, la ejecución continuará en el siguiente **case** si una instrucción **break** no está presente.

Puede tener **case** vacíos, como se muestra en este ejemplo:

```
switch(i) {
  case 1:
  case 2:
  case 3: System.out.println("i es 1, 2 o 3");
    break;
  case 4: System.out.println("i es 4");
    break;
}
```

En este fragmento, si **i** tiene el valor 1, 2 o 3, se ejecuta la primera instrucción **println()**. Si es 4, se ejecuta la segunda instrucción **println()**. El "apilamiento" de **case**, como se muestra en este ejemplo, es común cuando varios **case** comparten un código común.

Instrucciones switch anidadas

Es posible tener un **switch** como parte de la secuencia de instrucciones de un **switch** externo. A esto se le denomina **switch** anidado. Aun cuando las constantes de **case** de los **switch** interno y externo contengan valores comunes, no surgirán conflictos. Por ejemplo, el siguiente fragmento de código es perfectamente aceptable.

```
switch(ch1) {
  case 'A': System.out.println("Esta A es parte del switch externo.");
    switch(ch2) {
      case 'A':
        System.out.println("Esta A es parte del switch interno");
        break;
      case 'B': // ...
    } // fin del switch interno
    break;
  case 'B': // ...
```

Comprobación de avance

1. ¿De qué tipo puede ser la expresión que controla el **switch**?
2. Cuando la expresión **switch** coincide con una constante **case**, ¿qué sucede?
3. Si una secuencia **case** no termina en **break**, ¿qué sucede?

1. La expresión switch puede ser del tipo **char**, **short**, **int** o **byte**.
2. Cuando se encuentra una constante **case** coincidente, la secuencia de instrucciones asociada con ese **case** se ejecuta.
3. Si una secuencia **case** no termina con **break**, la ejecución continúa en la siguiente secuencia **case**, en caso de que exista una.

Proyecto 3.1 Empiece a construir un sistema de ayuda de Java

Ayuda.java

En este proyecto se construye un sistema simple de ayuda que despliega la sintaxis para las instrucciones de control de Java. El programa despliega un menú que contiene las instrucciones de control y luego espera a que usted elija una. Después de esto, se despliega la sintaxis de la instrucción. En esta primera versión del programa, la ayuda sólo está disponible para las instrucciones **if** y **switch**. Las instrucciones de control restantes se agregarán en proyectos posteriores.

Paso a paso

1. Cree un archivo llamado **Ayuda.java**.
2. El programa empieza por desplegar el siguiente menú:

```
Ayuda habilitada:
  1. if
  2. switch
Elija una:
```

Para lograr esto, deberá usar la secuencia de instrucciones que se muestra a continuación:

```
System.out.println("Ayuda habilitada:");
System.out.println("  1. if");
System.out.println("  2. switch");
System.out.print("Elija una: ");
```

3. A continuación, el programa obtiene la selección del usuario al llamar a **System.in.read()** tal y como aquí se muestra:

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

4. Una vez que se ha obtenido la selección, el programa usa la instrucción **switch** que se muestra aquí para desplegar la sintaxis de la instrucción seleccionada.

```
switch(inciso) {
  case '1':
    System.out.println("if:\n");
    System.out.println("if(condición) instrucción;");
    System.out.println("instrucción else;");
    break;
  case '2':
    System.out.println("switch:\n");
    System.out.println("switch(expresión) {");
    System.out.println("  constante case:");
    System.out.println("    secuencia de instrucciones");
    System.out.println("    break;");
    System.out.println("  // ...");
    System.out.println("}");
```

(continúa)

```
    break;
  default:
    System.out.print("Selección no encontrada.");
}
```

Observe cómo la cláusula **default** captura opciones no válidas. Por ejemplo, si el usuario ingresa 3, no habrá constantes **case** coincidentes, lo que ocasionará que se ejecute la secuencia **default**.

5. He aquí el listado completo del programa **Ayuda.java.**

```
/*
   Proyecto 3.1

   Un sistema simple de ayuda.
*/
class Ayuda {
  public static void main(String args[])
    throws java.io.IOException {
    char inciso;

    System.out.println("Ayuda habilitada:");
    System.out.println("  1. if");
    System.out.println("  2. switch");
    System.out.print("Elija una: ");
    inciso = (char) System.in.read();

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

    switch(inciso) {
      case '1':
        System.out.println("if:\n");
        System.out.println("if(condición) instrucción;");
        System.out.println("instrucción else;");
        break;
      case '2':
        System.out.println("switch:\n");
        System.out.println("switch(expresión) {");
        System.out.println("  constante case:");
        System.out.println("    secuencia de instrucciones");
        System.out.println("    break;");
        System.out.println("  // ...");
        System.out.println("}");
```

```
        break;
      default:
        System.out.print("Selección no encontrada.");
    }
  }
}
```

6. He aquí una ejecución de ejemplo:

```
Ayuda habilitada:
  1. if
  2. switch
Elija una: 1

if:

if(condición) instrucción;
instrucción else;
```

Pregunte al experto

P: **¿Bajo qué condiciones debe usar una escalera** if-else-if **en lugar de** switch **cuando está codificando un árbol de varias ramas?**

R: En general, usará una escalera **if-else-if** cuando las condiciones que controlan el proceso de selección no dependen de un solo valor. Por ejemplo, considere la siguiente secuencia **if-else-if**:

```
if(x < 10) // ...
else if(y != 0) // ...
else if(!terminado) // ...
```

Esta secuencia no puede recodificarse en **switch** porque las tres condiciones incluyen diferentes variables y tipos. ¿Cuál variable controlaría **switch**? Además, necesitará usar una escalera **if-else-if** cuando pruebe valores de punto flotante u otros objetos que no sean tipos válidos en una expresión **switch**.

HABILIDAD FUNDAMENTAL 3.4

El bucle for

Ha estado usando una forma simple del bucle **for** desde el módulo 1. Seguramente le sorprenderá lo poderoso y flexible que es el bucle **for**. Empecemos por revisar los fundamentos a partir de las formas más tradicionales de **for**.

La forma general del bucle **for** para repetir una sola instrucción es:

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

Para repetir un bloque, la forma general es

```
for(inicialización; condición; iteración)
{
 secuencia de instrucciones
}
```

La *inicialización* suele ser una instrucción de asignación que establece el valor inicial de la *variable de control del bucle*, el cual actúa como el contador que controla el bucle. La *condición* es una expresión boolean que determina si un bucle se repetirá o no. La expresión *iteración* define la cantidad a la que cambiará la variable de control del bucle cada vez que éste se repita. Observe que estas tres secciones principales del bucle deben estar separadas por puntos y comas. El bucle **for** se seguirá ejecutando siempre y cuando la condición se pruebe como verdadera. Una vez que la condición se vuelve falsa, el bucle se sale y la ejecución del programa se reinicia en la instrucción que sigue al **for**.

El siguiente programa usa un bucle **for** para imprimir las raíces cuadradas del 1 al 99. Asimismo, despliega el error de redondeo que está presente en cada raíz cuadrada.

```
// Muestra raíces cuadradas de 1 a 99 y error de redondeo.
class RaCuad {
  public static void main(String args[]) {
    double num, rcuad, erred;

    for(num = 1.0; num < 100.0; num++) {
      rcuad = Math.sqrt(num);
      System.out.println("La raíz cuadrada de " + num +
                         " es " + rcuad);

      // calcula el error de redondeo
      erred = num - (rcuad * rcuad);
      System.out.println("El error de redondeo es " + erred);
      System.out.println();
    }
  }
}
```

Observe que el error de redondeo se calcula al elevar al cuadrado la raíz cuadrada de cada número. Luego se resta este resultado al número original.

El bucle **for** puede avanzar de manera positiva o negativa, y puede cambiar la variable de control del bucle en cualquier cantidad. Por ejemplo, el siguiente programa imprime del 100 al –95 en decrementos de 5.

```
// Ejecución negativa de un bucle.
class DecrFor {
  public static void main(String args[]) {
    int x;

    for(x = 100; x > -100; x -= 5) ← La variable de control de bucle disminuye de 5 en 5.
      System.out.println(x);
  }
}
```

Un aspecto importante acerca de los bucles **for** es que la expresión condicional siempre se prueba en la parte superior del bucle. Esto significa que tal vez no se ejecute el código dentro del bucle si, para empezar, la condición es falsa. He aquí un ejemplo:

```
for(cuenta=10; cuenta < 5; cuenta++)
  x += cuenta; //esta instrucción no se ejecutará
```

Este bucle nunca se ejecutará porque su variable de control, **cuenta**, es mayor que 5 cuando se ingresa en el bucle. Esto hace que la expresión condicional, **cuenta < 5**, sea falsa desde el inicio; por lo tanto, ni siquiera una iteración del bucle tendrá lugar.

Algunas variaciones del bucle for

El bucle **for** es una de las instrucciones más versátiles en el lenguaje Java porque permite un amplio rango de variaciones. Por ejemplo, pueden usarse distintas variables de control del bucle. Considere el siguiente programa:

```
// Use comas en una instrucción for.
class Coma {
  public static void main(String args[]) {
    int i, j;

    for(i=0, j=10; i < j; i++, j--) ← Observe las dos variables de control del bucle.
      System.out.println("i y j: " + i + " " + j);


  }
}
```

A continuación se muestra la salida del programa:

```
i y j: 0 10
i y j: 1 9
i y j: 2 8
i y j: 3 7
i y j: 4 6
```

Aquí, las comas separan las dos instrucciones de inicialización y las dos expresiones de iteración. Cuando el bucle empieza, **i** y **j** se inicializan. Cada vez que el bucle se repite, **i** se incrementa y **j** se reduce. A menudo es conveniente usar varias variables de control del bucle porque pueden simplificar ciertos algoritmos. Es posible tener cualquier número de instrucciones de inicialización e iteración, pero en la práctica más de dos o tres hacen que el bucle **for** resulte difícil de manejar.

La condición que controla el bucle puede ser cualquier expresión booleana válida. No es necesario que se incluya la variable de control del bucle. En el siguiente ejemplo, el bucle sigue ejecutándose hasta que el usuario escribe la letra S en el teclado.

```
// Bucle hasta que se escribe S.
class PruebaFor {
  public static void main(String args[])
    throws java.io.IOException {

    int i;

    System.out.println("Oprima S para detener.");

    for(i = 0; (char) System.in.read() != 'S'; i++)
      System.out.println("Paso #" + i);
  }
}
```

Piezas faltantes

Algunas variaciones interesantes del bucle **for** se crean al dejar vacías las piezas de la definición del bucle. En Java, es posible que cualquiera o todas las inicializaciones, condiciones o partes de iteración del bucle **for**, estén en blanco. Revise, por ejemplo, el siguiente programa:// Partes de **for** pueden estar en blanco.

```
Partes de for pueden estar vacías.
class Vac {
  public static void main(String args[]) {
    int i;

    for(i = 0; i < 10; ) {          <-- Falta la expresión de iteración.
      System.out.println("Paso #" + i);
```

```
      i++; // incrementa la var de control de bucle
    }
  }
}
```

Aquí, la expresión de iteración de **for** está vacía. En cambio, la variable de control del bucle **i** aumenta dentro del cuerpo del bucle. Esto significa que cada vez que el bucle se repite, se prueba **i** para comprobar si es igual a 10, pero ninguna otra acción se toma. Por supuesto, como **i** aún se incrementa dentro del cuerpo del bucle, el bucle se ejecuta normalmente desplegando la siguiente salida:

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

En el siguiente ejemplo, la parte de la inicialización se mueve también afuera del **for**.

```
// Mover afuera del bucle for.
class Vac2 {
  public static void main(String args[]) {
    int i;

    i = 0; // saca la inicialización del bucle
    for(; i < 10; ) {
      System.out.println("Paso #" + i);
      i++; // incrementa la var de control del bucle
    }
  }
}
```

La expresión de inicialización se saca del bucle.

En esta versión, en lugar de hacerlo como parte de **for**, **i** se inicializa antes de que el bucle inicie. Por lo general, querrá inicializar la variable de control del bucle dentro de **for**. La inicialización solamente suele colocarse fuera del bucle cuando se deriva el valor inicial mediante un proceso complejo que no es posible incluir dentro de la instrucción **for**.

El bucle infinito

Puede crear un *bucle infinito* (un bucle que nunca termina) empleando el **for** y dejando vacía la expresión condicional. Por ejemplo, el siguiente fragmento muestra la manera en que la mayoría de los programadores de Java crean un bucle infinito.

```
for(;;) // bucle intencionalmente infinito
{
  // ...
}
```

Este bucle se ejecutará para siempre. Aunque hay algunas tareas de programación, como procesadores de comando del sistema operativo que requieren un bucle infinito, la mayor parte de los "bucles infinitos" tan sólo son bucles con requisitos especiales de terminación. Casi al final de este módulo verá cómo detener un bucle de este tipo. (Sugerencia: se hace empleando la instrucción **break**.)

Bucles sin cuerpo

En Java, el cuerpo asociado con un bucle **for** (o cualquier otro bucle) puede estar vacío. Esto se debe a que una *instrucción null* es sintácticamente válida. Los bucles sin cuerpo suelen resultar útiles. Por ejemplo, el siguiente programa usa uno para sumar del 1 al 5.

```
// El cuerpo de un bucle puede estar vacío.
class Vac3 {
  public static void main(String args[]) {
    int i;
    int suma = 0;

    // suma los números hasta 5
    for(i = 1; i <= 5; suma += i++) ;   <-- ¡No hay cuerpo en este bucle!

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

Aquí se muestra la salida del programa:

```
La suma es 15
```

Observe que el proceso de suma se maneja por completo dentro de la instrucción **for** y no se necesita el cuerpo. Ponga especial atención a la expresión de iteración:

```
suma += i++
```

No se intimide con instrucciones como éstas. Son comunes en programa de Java escritos profesionalmente y son fáciles de comprender si las divide en partes. En otras palabras, esta instrucción dice "agrega a **suma** el valor de **suma** más **i**, y luego aumenta **i**". Por lo tanto, es lo mismo que la secuencia de instrucciones:

```
suma = suma + i;
i++;
```

Declaración de las variables de control del bucle dentro del bucle for

A menudo la variable que controla un bucle **for** sólo es necesaria para el bucle por lo que no se utiliza en ningún otro lado. Cuando éste es el caso, es posible declarar la variable dentro de la parte de inicialización del **for**. Por ejemplo, el siguiente programa calcula la sumatoria y el factorial del 1 al 5 y declara su variable de control de bucle **i** dentro del **for**.

```
// Declara la variable de control del bucle dentro de for.
class ForVar {
  public static void main(String arg s[]) {
    int suma = 0;
    int fact = 1;

    // calcula el factorial de los números hasta el 5
    for(int i = 1; i <= 5; i++) {
      suma += i;  // i se conoce en todo el bucle
      fact *= i;
    }

    // pero i no se conoce aquí.

    System.out.println("La suma es " + suma);
    System.out.println("El factorial es " + fact);
  }
}
```

La variable **i** se declara dentro de la instrucción **for**.

Cuando se declara una variable dentro de un bucle **for**, hay que recordar algo importante: el alcance de la variable termina cuando la instrucción **for** termina (es decir, el alcance de la variable está limitado por el bucle **for**). Fuera de éste, la variable dejará de existir. Por consiguiente, en el ejemplo anterior, **i** no es accesible fuera del bucle **for**. Si necesita usar la variable de control del bucle en otro lugar del programa, no podrá declararla dentro del bucle **for**.

Antes de seguir adelante, tal vez quiera experimentar sus propias variaciones en el bucle **for**. Como lo verá, es un bucle fascinante.

El bucle for mejorado

Recientemente, una nueva forma del bucle **for**, llamada ***for** mejorado*, se agregó a Java. El **for** mejorado proporciona una forma mejorada de recorrer a manera de ciclo el contenido de una colección de objetos como, por ejemplo, una matriz. El bucle **for** mejorado se analizará en el capítulo 5 después de que se hayan introducido las matrices.

Comprobación de avance

1. ¿Parte de una instrucción **for** puede estar vacía?
2. Muestre cómo crear un bucle infinito empleando **for**.
3. ¿Cuál es el alcance de una variable declarada dentro de una instrucción **for**?

HABILIDAD FUNDAMENTAL 3.5

El bucle while

Otro bucle de Java es el **while**. La forma general del bucle **while** es

while(*condición*) *instrucción*;

donde *instrucción* puede ser una sola instrucción o un bloque de instrucciones, y *condición* define la condición que controla el bucle y puede ser cualquier expresión booleana válida. El bucle se repite mientras la condición sea verdadera. Cuando la condición se vuelve falsa, el control del programa pasa a la línea que le sigue inmediatamente en el bucle.

He aquí un ejemplo simple en el que un **while** se utiliza para imprimir el alfabeto:

```
// Demuestra el bucle while.
class WhileDemo {
  public static void main(String args[]) {
    char ch;

    // imprime el alfabeto con el bucle while
    ch = 'a';
    while(ch <= 'z') {
```

1. Sí, las tres partes del **for** (inicialización, condición e iteración) pueden estar vacías.
2. for(;;)
3. El alcance de una variable declarada dentro de un **for** está limitado al bucle. Fuera de él, no se le conoce.

```
      System.out.print(ch);
      ch++;
    }

  }

}
```

Aquí, **ch** se inicializa en la letra a. Cada vez que se recorre el bucle, se da salida a **ch** y luego se incrementa. Este proceso sigue hasta que **ch** llega a z.

Al igual que el bucle **for**, el **while** revisa la expresión condicional en la parte superior del bucle, lo que significa que es posible que el código del bucle no se ejecute. Esto elimina la necesidad de realizar una prueba aparte antes de recorrer el bucle. El siguiente programa ilustra esta característica del bucle **while** y calcula las potencias enteras de 2, desde 0 hasta 9.

```
// Calcula potencias enteras de 2.
class Potencias {
  public static void main(String args[]) {
    int e;
    int result;

    for(int i=0; i < 10; i++) {
      result = 1;
      e = i;
      while(e > 0) {
        result *= 2;
        e--;
      }

      System.out.println("2 a la " + i +
                         " potencia es " + result);
    }
  }
}
```

Aquí se muestra la salida del programa:

```
2 a la 0 potencia es 1
2 a la 1 potencia es 2
2 a la 2 potencia es 4
2 a la 3 potencia es 8
2 a la 4 potencia es 16
2 a la 5 potencia es 32
2 a la 6 potencia es 64
2 a la 7 potencia es 128
2 a la 8 potencia es 256
2 a la 9 potencia es 512
```

Pregunte al experto

P: **Dada la flexibilidad inherente a todos los bucles de Java, ¿cuáles criterios deben considerarse cuando se selecciona un bucle? Es decir, ¿cómo elijo el bucle correcto para un trabajo específico?**

R: Use un bucle **for** cuando realice un número conocido de iteraciones. Use **do-while** cuando necesite un bucle que siempre realice al menos una iteración. El **while** se usa mejor cuando el bucle se repite un número desconocido de veces.

Observe que el bucle **while** sólo se ejecuta cuando **e** es mayor que 0. Por lo tanto, cuando **e** es cero, como en la primera iteración del bucle **for**, el bucle **while** se omite.

HABILIDAD FUNDAMENTAL 3.6

El bucle do-while

El último de los bucles de Java es el **do-while**. A diferencia de los bucles **for** y **while**, en los que la condición se prueba en la parte superior del bucle, el bucle **do-while** revisa su condición en la parte inferior. Esto significa que un bucle **do-while** siempre se ejecutará por lo menos una vez. La forma general del bucle **do-while** es

```
do {
  instrucciones;
}while(condición);
```

Aunque las llaves son necesarias cuando sólo una instrucción está presente, a menudo se utilizan para mejorar la legibilidad del constructor **do-while** evitando así la confusión con el **while**. El bucle **do-while** se ejecuta siempre y cuando la expresión condicional sea verdadera.

El siguiente programa recorre en bucle hasta que el usuario ingresa la letra q.

```
// Demuestra el bucle do-while.
class DWDemo {
  public static void main(String args[])
    throws java.io.IOException {

    char ch;

    do {
      System.out.print("Oprima una tecla y luego ENTER: ");
```

```
      ch = (char) System.in.read(); // obtiene un carácter
    } while(ch != 'q');
  }
}
```

Empleando un bucle **do-while** podemos mejorar aún más el programa de juego de adivinanza del principio de este módulo. Esta vez, el programa recorrerá en bucle hasta que adivine la letra.

```
// Juego de adivinar la letra, 4a versión.
class Adiv4 {
  public static void main(String args[])
    throws java.io.IOException {

    char ch, respuesta = 'K';

    do {
    System.out.println("Estoy pensando en una letra entre la A y la Z.");
    System.out.print("Puedes adivinarla: ");

      // lee una letra, pero omite retornos de carro
      do {
        ch = (char) System.in.read(); // obtiene un carácter
      } while(ch == '\n' | ch == '\r');

      if(ch == respuesta) System.out.println("** Correcto **");
      else {
        System.out.print("...Lo siento, te encuentras ");
        if(ch < respuesta) System.out.println("demasiado bajo");
        else System.out.println("demasiado alto");
        System.out.println("Prueba de nuevo\n");
      }
    } while(respuesta != ch);
  }
}
```

He aquí una ejecución de ejemplo:

```
Estoy pensando en una letra entre la A y la Z.
Puedes adivinarla: A
...Lo siento, te encuentras demasiado bajo
Prueba de nuevo
```

```
Estoy pensando en una letra entre la A y la Z.
Puedes adivinarla: Z
...Lo siento, te encuentras demasiado alto
Prueba de nuevo

Estoy pensando en una letra entre la A y la Z.
Puedes adivinarla: K
** Correcto **
```

Observe otro elemento interesante en este programa. El bucle **do-while** mostrado aquí obtiene el siguiente carácter omitiendo cualquier carácter de retorno de carro que pudiera estar en el flujo de entrada:

```
// lee una letra, pero omite retornos de carro
      do {
        ch = (char) System.in.read(); // obtiene un carácter
      } while(ch == '\n' | ch == '\r');
```

He aquí por qué es necesario el bucle: como se explicó antes, **System.in** almacena en cola los caracteres: usted debe oprimir ENTER antes de enviar los caracteres. Al oprimir ENTER se genera un retorno de carro. Este carácter permanece pendiente en la memoria de entrada. Este bucle descarta estos caracteres y sigue leyendo la entrada hasta que ya no queda ningún carácter presente.

Comprobación de avance

1. ¿Cuál es la principal diferencia entre los bucles **while** y **do-while**?

2. La condición que controla el **while** puede ser de cualquier tipo. ¿Cierto o falso?

1. El **while** revisa su condición en la parte superior del bucle. El **do-while** la revisa en la parte inferior.
2. Falso. La condición debe ser del tipo **boolean**.

Proyecto 3.2 Mejore el sistema de ayuda de Java

Ayuda2.java

En este proyecto se expande el sistema de ayuda de Java que se creó en el proyecto 3.1. Esta versión agrega la sintaxis para los bucles **for**, **while** y **do-while**. Asimismo, revisa la selección de menú del usuario recorriendo en bucle hasta que se ingresa una respuesta válida.

Paso a paso

1. Copie **Ayuda.java** en un nuevo archivo llamado **Ayuda2.java**.
2. Cambie la parte del programa que despliega las opciones con el fin de que éste use el bucle mostrado aquí:

```
do {
  System.out.println("Ayuda habilitada:");
  System.out.println("  1. if");
  System.out.println("  2. switch");
  System.out.println("  3. for");
  System.out.println("  4. while");
  System.out.println("  5. do-while\n");
  System.out.print("Elija una: ");
  do {
    inciso = (char) System.in.read();
  } while(inciso == '\n' | inciso == '\r');
} while( inciso < '1' | inciso > '5');
```

   Observe que un bucle **do-while** anidado se utiliza para descartar cualquier carácter de retorno de carro que pudiera estar presente en el flujo de entrada. Después de hacer este cambio, el programa se recorrerá en bucle desplegando el menú hasta que el usuario ingrese una respuesta que se encuentre entre 1 y 5.

3. Expanda la instrucción **switch** para que incluya los bucles **for**, **while** y **do-while** como se muestra aquí:

```
switch(inciso) {
  case '1':
    System.out.println("if:\n");
    System.out.println("if(condición) instrucción;");
    System.out.println("instrucción else;");
    break;
```

(continúa)

```
  case '2':
    System.out.println("switch:\n");
    System.out.println("switch(expresión) {");
    System.out.println("  constante case:");
    System.out.println("    secuencia de instrucciones");
    System.out.println("    break;");
    System.out.println("  // ...");
    System.out.println("}");
    break;
  case '3':
    System.out.println("for:\n");
    System.out.print("for(inic; condición; iteración)");
    System.out.println(" instrucción;");
    break;
  case '4':
    System.out.println("while:\n");
    System.out.println("while(condición) instrucción;");
    break;
  case '5':
    System.out.println("do-while:\n");
    System.out.println("do {");
    System.out.println("  instrucción;");
    System.out.println("} while (condición);");
    break;
}
```

Observe que ninguna instrucción **default** está presente en esta versión de **switch**. Debido a que el bucle de menú asegura que se introducirá una respuesta válida, ya no es necesario incluir una instrucción **default** para manejar una opción no válida.

4. He aquí el listado completo del programa **Ayuda2.java**:

```
/*
   Proyecto 3.2

   Un sistema mejorado de ayuda que usa
   do-while para procesar una selección de menú.
*/
class Ayuda2 {
  public static void main(String args[])
    throws java.io.IOException {
    char inciso;

    do {
      System.out.println("Ayuda habilitada:");
```

```
      System.out.println("  1. if");
      System.out.println("  2. switch");
      System.out.println("  3. for");
      System.out.println("  4. while");
      System.out.println("  5. do-while\n");
      System.out.print("Elija una: ");
      do {
        inciso = (char) System.in.read();
      } while(inciso == '\n' | inciso == '\r');
    } while( inciso < '1' | inciso > '5');

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

    switch(inciso) {
      case '1':
        System.out.println("if:\n");
        System.out.println("if(condición) instrucción;");
        System.out.println("instrucción else;");
        break;
      case '2':
        System.out.println("switch:\n");
        System.out.println("switch(expresión) {");
        System.out.println("  constante case:");
        System.out.println("    secuencia de instrucciones");
        System.out.println("    break;");
        System.out.println("  // ...");
        System.out.println("}");
        break;
      case '3':
        System.out.println("for:\n");
        System.out.print("for(inic; condición; iteración)");
        System.out.println(" instrucción;");
        break;
      case '4':
        System.out.println("while:\n");
        System.out.println("while(condición) instrucción;");
        break;
      case '5':
        System.out.println("do-while:\n");
        System.out.println("do {");
        System.out.println("  instrucción;");
        System.out.println("} while (condición);");
        break;
    }
  }
}
```

HABILIDAD FUNDAMENTAL 3.7

Uso de break para salir de un bucle

Es posible forzar la salida inmediata de un bucle omitiendo cualquier código en el cuerpo del bucle, así como la prueba de condición del bucle empleando la instrucción **break**. Cuando se encuentra una instrucción **break** dentro de un bucle, éste se termina y el control del programa se reanuda en la siguiente instrucción del bucle. He aquí un ejemplo simple:

```
// Uso de break para salir de un bucle.
class BreakDemo {
  public static void main(String args[]) {
    int num;

    num = 100;

    // recorre el bucle si i al cuadrado es menor que num
    for(int i=0; i < num; i++) {
      if(i*i >= num) break; // termina el bucle si i*i >= 100
      System.out.print(i + " ");
    }
    System.out.println("Bucle completo.");
  }
}
```

Este programa genera la siguiente salida:

```
0 1 2 3 4 5 6 7 8 9 Bucle completo.
```

Como verá, aunque el bucle **for** está diseñado para ejecutar de 0 a **num** (que en este caso es 100), la instrucción **break** hace que finalice antes cuando i al cuadrado es mayor que **num**.

La instrucción **break** puede usarse con cualquier bucle de Java, incluyendo bucles intencionalmente infinitos. Por ejemplo, el siguiente programa simplemente lee la entrada hasta que el usuario ingresa la letra q.

```
// Lee la entrada hasta que se recibe una q.
class Break2 {
  public static void main(String args[])
    throws java.io.IOException {

    char ch;

    for( ; ; ) {
      ch = (char) System.in.read(); // obtiene un carácter
      if(ch == 'q') break;
```

El bucle "infinito" termina con el **break**.

```
    }
    System.out.println("Introdujo una q!");
  }
}
```

Cuando se usa dentro de un conjunto anidado de bucles, la instrucción **break** sólo sale del bucle más interno. Por ejemplo:

```
// Uso de break con bucles anidados.
class Break3 {
  public static void main(String args[]) {

    for(int i=0; i<3; i++) {
      System.out.println("Cuenta del bucle externo: " + i);
      System.out.print("    Cuenta del bucle interno: ");

      int t = 0;
      while(t < 100) {
        if(t == 10) break; // termina el bucle si t es 10
        System.out.print(t + " ");
        t++;
      }
      System.out.println();
    }
    System.out.println("Bucles completos.");
  }
}
```

Este programa genera la siguiente salida:

```
Cuenta del bucle externo: 0
    Cuenta del bucle interno: 0 1 2 3 4 5 6 7 8 9
Cuenta del bucle externo: 1
    Cuenta del bucle interno: 0 1 2 3 4 5 6 7 8 9
Cuenta del bucle externo: 2
    Cuenta del bucle interno: 0 1 2 3 4 5 6 7 8 9
Bucles completos.
```

Como puede ver, la instrucción **break** en el bucle interno sólo causa la terminación de ese bucle. El bucle externo no se ve afectado.

He aquí dos conceptos que se deben recordar acerca de **break**. En primer lugar, puede aparecer más de una instrucción **break** en un bucle; sin embargo, tenga cuidado pues demasiadas instrucciones **break** tienden a desestructurar su código. En segundo lugar, el **break** que termina una instrucción **switch** sólo afecta a esta instrucción **switch** y no a cualquier bucle incluido.

HABILIDAD FUNDAMENTAL 3.8

Use break como una forma de goto

Además de su uso con la instrucción **switch** y con los bucles, la instrucción **break** puede emplearse por sí sola para proporcionar una forma "civilizada" de la instrucción goto. Java no cuenta con una instrucción goto porque ésta proporciona una manera no estructurada de modificar el flujo de ejecución del programa. Los programas que llevan a cabo un uso extenso de goto suelen ser difíciles de comprender y de mantener. Sin embargo, existen ciertos lugares en los que goto resulta un dispositivo útil y legítimo. Por ejemplo, goto puede ser útil al salir de conjuntos profundamente anidados de bucles. Para manejar estas situaciones, Java define una forma expandida de la instrucción **break**. Al usar esta forma de **break**, puede salir de uno o más bloques de código. No es necesario que estos bloques sean parte de un bloque o un **switch**; pueden ser cualquier bloque. Más aún, usted puede especificar con precisión donde se reanudará la ejecución pues esta forma de **break** funciona con una etiqueta. Como verá, **break** le proporciona los beneficios de un goto sin que padezca los problemas propios de este último.

La forma general de la instrucción **break** etiquetada se muestra a continuación:

break *etiqueta*;

Aquí, *etiqueta* es el nombre de una etiqueta que identifica un bloque de código. Cuando se ejecuta esta forma de **break**, el control se transfiere fuera del bloque de código con nombre. El bloque etiquetado de código debe encerrar la instrucción **break**, pero no es necesario que sea el bloque inmediato. Esto significa que usted puede usar una instrucción **break** etiquetada para salir de un conjunto de bloques anidados. Sin embargo, no puede usar **break** para transferir el control a un bloque de código que no contiene la instrucción **break**.

Para asignar un nombre a un bloque, póngale una etiqueta al inicio. El bloque que se etiqueta puede ser un bloque independiente o una instrucción que tenga un bloque como destino. Una *etiqueta* es cualquier identificador válido de Java seguido por un punto y coma. Una vez que haya etiquetado un bloque, puede usar esta etiqueta como destino de una instrucción **break**. Al hacerlo, hará que la ejecución se reanude al *final* del bloque etiquetado. Por ejemplo, el siguiente programa muestra tres bloques anidados:

```
// Uso de break con una etiqueta.
class Break4 {
  public static void main(String args[]) {
    int i;

    for(i=1; i<4; i++) {
uno:  {
dos:    {
tres:    {
            System.out.println("\ni es " + i);
```

```
            if(i==1) break uno;      ← Salida a una etiqueta.
            if(i==2) break dos;
            if(i==3) break tres;

            // este nunca se alcanza
            System.out.println("no se imprime");
          }
          System.out.println("Tras el bloque tres.");
        }
        System.out.println("Tras el bloque dos.");
      }
      System.out.println("Tras el bloque uno.");
    }
    System.out.println("Tras for.");

  }
}
```

Aquí se muestra la salida del programa:

```
i es 1
Tras el bloque uno.

i es 2
Tras el bloque dos.
Tras el bloque uno.

i es 3
Tras el bloque tres.
Tras el bloque dos.
Tras el bloque uno.
Tras for.
```

Revisemos detalladamente el programa para comprender de manera precisa por qué se produjo esta salida. Cuando **i** es 1, la primera instrucción **if** tiene éxito, lo que ocasiona que un **break** termine el bloque de código definido por la etiqueta **uno**. Esto hace que **Tras el bloque uno** se imprima. Cuando **i** es 2, el segundo **if** tiene éxito, lo que ocasiona que el control se transfiera al final del bloque etiquetado como **dos**. Esto hace que se impriman los mensajes **Tras el bloque dos.** y **Tras el bloque uno.**, en ese orden. Cuando **i** es 3, la tercera instrucción **if** tiene éxito y el control se transfiere al final del bloque etiquetado como **tres**. En ese momento, se despliegan los tres mensajes.

He aquí otro ejemplo. Esta vez **break** se emplea para saltar afuera de una serie de bucles **for** anidados. Cuando se ejecuta la instrucción **break** en el bucle interno, el control del programa salta al

final del bloque definido por el bucle **for** externo, el cual está etiquetado como **hecho**. Esto hace que el resto de los tres bucles se omita.

```
// Otro ejemplo del uso de break con una etiqueta.
class Break5 {
  public static void main(String args[]) {

hecho:
    for(int i=0; i<10; i++) {
      for(int j=0; j<10; j++) {
        for(int k=0; k<10; k++) {
          System.out.println(k + " ");
          if(k == 5) break hecho; // salta a hecho
        }
        System.out.println("Tras bucle k"); // no se ejecuta
      }
      System.out.println("Tras bucle j"); // no se ejecuta
    }
    System.out.println("Tras bucle i");
  }
}
```

Aquí se muestra la salida del programa:

```
0
1
2
3
4
5
Tras bucle i
```

El lugar preciso en que pone una etiqueta es muy importante, sobre todo cuando se trabaja con bucles. Por ejemplo, considere el siguiente programa:

```
// El lugar donde pone una etiqueta es muy importante.
class Break6 {
  public static void main(String args[]) {
    int x=0, y=0;

// aquí, ponga la etiqueta antes de la instrucción for.
alto1: for(x=0; x < 5; x++) {
         for(y = 0; y < 5; y++) {
```

```
            if(y == 2) break alto1;
            System.out.println("x y y: " + x + " " + y);
          }
        }

        System.out.println();

// ahora, ponga la etiqueta inmediatamente antes {
      for(x=0; x < 5; x++)
alto2: {
          for(y = 0; y < 5; y++) {
            if(y == 2) break alto2;
            System.out.println("x y y: " + x + " " + y);
          }
        }

  }
}
```

Aquí se muestra la salida del programa:

```
x y y: 0 0
x y y: 0 1

x y y: 0 0
x y y: 0 1
x y y: 1 0
x y y: 1 1
x y y: 2 0
x y y: 2 1
x y y: 3 0
x y y: 3 1
x y y: 4 0
x y y: 4 1
```

En el programa ambos conjuntos de bucles anidados son iguales, excepto en un aspecto. En el primer conjunto, la etiqueta avanza desde el bucle **for** externo. En este caso, cuando se ejecuta **break**, el control se transfiere al final de todo el bloque **for** omitiendo el resto de las iteraciones del bucle externo. En el segundo conjunto, la etiqueta procede de las llaves de apertura de **for**. Por lo tanto, cuando se ejecuta **break alto2**, el control se transfiere al final del bloque **for** externo, lo que ocasiona que la siguiente iteración ocurra.

Tenga en cuenta que no puede salir alguna etiqueta que no esté definida para un bloque incluyente. Por ejemplo, el siguiente programa no es válido y no se compilará.

```
// Este programa contiene un error.
class BreakErr {
  public static void main(String args[]) {

    uno: for(int i=0; i<3; i++) {
      System.out.print("Pasa " + i + ": ");
    }

    for(int j=0; j<100; j++) {
      if(j == 10) break uno; // EQUIVOCADO
      System.out.print(j + " ");
    }
  }
}
```

Debido a que el bucle etiquetado como **uno** no contiene la instrucción **break**, no es posible transferir el control a ese bloque.

Pregunte al experto

P: **Usted dice que** goto **no está estructurado y que el break con una etiqueta ofrece una mejor opción. Pero, en realidad, ¿dejar de salir a una etiqueta, lo que podría eliminar muchas líneas de código y niveles de anidado del** break**, no podría también acabar con la estructura del código?**

R: ¡La respuesta corta es sí! Sin embargo, en esos casos en los que se requiere un cambio en el flujo del programa, salir a una etiqueta retiene aún cierta estructura, ¡**goto** no tiene ninguna!

HABILIDAD FUNDAMENTAL 3.9

El uso de continue

Es posible forzar una iteración temprana de un bucle omitiendo la estructura de control normal del bucle. Esto se logra usando la instrucción **continue**, la cual impone la ejecución de la siguiente iteración del bucle omitiendo cualquier código entre sí mismo y la expresión condicional que controla el bucle. Por lo tanto, **continue** es esencialmente el complemento de **break**. Por ejemplo, el siguiente programa usa **continue** como ayuda para imprimir los números pares entre el 0 y el 100.

```
// Uso de continue.
class ContDemo {
  public static void main(String args[]) {
    int i;

    // imprime números pares entre 0 y 100
    for(i = 0; i<=100; i++) {
      if((i%2) != 0) continue; // itera
      System.out.println(i);
    }
  }
}
```

Sólo se imprimen los números pares porque uno no causaría que el bucle se iterara antes, omitiendo la llamada a **println()**.

En los bucles **while** y **do-while**, una instrucción **continue** hará que el control vaya directamente a la expresión condicional y luego siga el recorrido del bucle. En el caso de **for**, se evalúa la expresión de iteración del bucle, luego se ejecuta la expresión condicional y más tarde se sigue el bucle.

Al igual que con la instrucción **break**, **continue** puede especificar una etiqueta para describir en cuál bucle incluido se deberá continuar. A continuación se presenta un programa de ejemplo que usa **continue** con una etiqueta.

```
// Uso de continue con una etiqueta.
class ContAEtiqueta {
  public static void main(String args[]) {

bucleexterno:
    for(int i=1; i < 10; i++) {
      System.out.print("\nPaso de bucle externo " + i +
                       ", Bucle interno: ");
      for(int j = 1; j < 10; j++) {
        if(j == 5) continue bucleexterno; // continua bucle externo
        System.out.print(j);
      }
    }
  }
}
```

Aquí se muestra la salida del programa:

```
Paso de bucle externo 1, Bucle interno: 1234
Paso de bucle externo 2, Bucle interno: 1234
```

```
Paso de bucle externo 3, Bucle interno: 1234
Paso de bucle externo 4, Bucle interno: 1234
Paso de bucle externo 5, Bucle interno: 1234
Paso de bucle externo 6, Bucle interno: 1234
Paso de bucle externo 7, Bucle interno: 1234
Paso de bucle externo 8, Bucle interno: 1234
Paso de bucle externo 9, Bucle interno: 1234
```

Como lo muestra la salida, cuando se ejecuta **continue**, el control pasa al bucle externo omitiendo el resto del bucle interno.

El buen uso de **continue** resulta raro. Una razón es que Java proporciona un rico conjunto de instrucciones de bucle que se amoldan a la mayor parte de las aplicaciones. Sin embargo, para esas circunstancias especiales en las que se necesita una iteración temprana, la instrucción **continue** proporciona una manera estructurada de lograrlo.

Comprobación de avance

1. Dentro de un bucle, ¿qué sucede cuando se ejecuta un **break** (sin etiqueta)?
2. ¿Qué sucede cuando se ejecuta un **break** con una etiqueta?
3. ¿Qué hace **continue**?

1. Dentro de un bucle, un **break** sin una etiqueta causa la terminación inmediata del bucle. La ejecución se reanuda en la primera línea del código después del bucle.
2. Cuando se ejecuta un **break** etiquetado, la ejecución se reanuda al final del bloque etiquetado.
3. La instrucción **continue** hace que un bucle se itere de inmediato, omitiendo cualquier código restante. Si **continue** incluye una etiqueta, el bucle etiquetado continúa.

Proyecto 3.3 Termine el sistema de ayuda de Java

Ayuda3.java

Este proyecto pone los toques finales en el sistema de ayuda de Java que se creó en los proyectos anteriores. Esta versión agrega la sintaxis para **break** y **continue**. También permite al usuario solicitar la sintaxis para más de una oración. Esto lo realiza agregando un bucle externo que se ejecuta hasta que el usuario introduce **q** como selección de menú.

Paso a paso

1. Copie **Ayuda2.java** en un nuevo archivo llamado **Ayuda3.java**.
2. Rodee todo el código del programa con un bucle **for** infinito. Salga de este bucle, empleando **break**, cuando se ingrese una letra **q**. Como este bucle rodea todo el código del programa, la salida de dicho bucle hace que el programa termine.
3. Cambie el bucle del menú como se muestra aquí:

```
do {
  System.out.println("Ayuda habilitada:");
  System.out.println("  1. if");
  System.out.println("  2. switch");
  System.out.println("  3. for");
  System.out.println("  4. while");
  System.out.println("  5. do-while");
  System.out.println("  6. break");
  System.out.println("  7. continue\n");
  System.out.print("Elija una (q para salir): ");
  do {
    inciso = (char) System.in.read();
  } while(inciso == '\n' | inciso == '\r');
} while( inciso < '1' | inciso > '7' & inciso != 'q');
```

Observe que este bucle incluye ahora las instrucciones **break** y **continue**. También acepta la letra **q** como opción válida.

4. Expanda la instrucción **switch** para incluir las instrucciones **break** y **continue**, como se muestra aquí.

```
case '6':
  System.out.println("break:\n");
  System.out.println("break; o break etiqueta;");
  break;
case '7':
  System.out.println("continue:\n");
  System.out.println("continue; o continue etiqueta;");
  break;
```

(continúa)

5. He aquí el listado completo de **Ayuda3.java**:

```
/*
   Proyecto 3.3

   El sistema terminado de ayuda de instrucciones de Java
   que procesa varias solicitudes.
*/
class Ayuda3 {
  public static void main(String args[])
    throws java.io.IOException {
    char inciso;

    for(;;) {
      do {
        System.out.println("Ayuda habilitada:");
        System.out.println("  1. if");
        System.out.println("  2. switch");
        System.out.println("  3. for");
        System.out.println("  4. while");
        System.out.println("  5. do-while");
        System.out.println("  6. break");
        System.out.println("  7. continue\n");
        System.out.print("Elija una (q para salir): ");
        do {
          inciso = (char) System.in.read();
        } while(inciso == '\n' | inciso == '\r');
      } while( inciso < '1' | inciso > '7' & inciso != 'q');

      if(inciso == 'q') break;

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

      switch(inciso) {
        case '1':
          System.out.println("if:\n");
          System.out.println("if(condición) instrucción;");
          System.out.println("instrucción else;");
          break;
        case '2':
          System.out.println("switch:\n");
          System.out.println("switch(expresión) {");
          System.out.println("  constante case:");
          System.out.println("    secuencia de instrucciones");
          System.out.println("    break;");
          System.out.println("  // ...");
          System.out.println("}");
          break;
```

```
        case '3':
          System.out.println("for:\n");
          System.out.print("for(inic; condición; iteración)");
          System.out.println(" instrucción;");
          break;
        case '4':
          System.out.println("while:\n");
          System.out.println("while(condiciónn) instrucción;");
          break;
        case '5':
          System.out.println("do-while:\n");
          System.out.println("do {");
          System.out.println("  instrucción;");
          System.out.println("} while (condición);");
          break;
        case '6':
          System.out.println("break:\n");
          System.out.println("break; o break etiqueta;");
          break;
        case '7':
          System.out.println("continue:\n");
          System.out.println("continue; o continue etiqueta;");
          break;
      }
      System.out.println();
    }
  }
}
```

6. He aquí una ejecución de ejemplo:

```
Ayuda habilitada:
  1. if
  2. switch
  3. for
  4. while
  5. do-while
  6. break
  7. continue

Elija una (q para salir): 1

if:

if(condición) instrucción;
instrucción else;

Ayuda habilitada:
```

(continúa)

```
  1. if
  2. switch
  3. for
  4. while
  5. do-while
  6. break
  7. continue

Elija una (q para salir): 6

break:

break; o break etiqueta;

Ayuda habilitada:
  1. if
  2. switch
  3. for
  4. while
  5. do-while
  6. break
  7. continue

Elija una (q para salir): q
```

HABILIDAD FUNDAMENTAL 3.10

Bucles anidados

Como ha visto en algunos de los ejemplos anteriores, es posible anidar un bucle dentro de otro. Los bucles anidados se usan para resolver una amplia variedad de problemas de programación, así que son parte esencial de ésta. Antes de abandonar el tema de las instrucciones de bucle de Java, veamos un ejemplo más de bucle anidado. El siguiente programa usa un bucle **for** anidado para encontrar los factores del 2 al 100.

```
/*
   Uso de bucles anidados para encontrar factores
   de números entre 2 y 100.
*/
class EncFac {
  public static void main(String args[]) {

    for(int i=2; i <= 100; i++) {
      System.out.print("Factores de " + i + ": ");
      for(int j = 2; j < i; j++)
        if((i%j) == 0) System.out.print(j + " ");
      System.out.println();
    }
```

```
  }
}
```

He aquí una parte de la salida producida por el programa:

```
Factores de 2:
Factores de 3:
Factores de 4: 2
Factores de 5:
Factores de 6: 2 3
Factores de 7:
Factores de 8: 2 4
Factores de 9: 3
Factores de 10: 2 5
Factores de 11:
Factores de 12: 2 3 4 6
Factores de 13:
Factores de 14: 2 7
Factores de 15: 3 5
Factores de 16: 2 4 8
Factores de 17:
Factores de 18: 2 3 6 9
Factores de 19:
Factores de 20: 2 4 5 10
```

En el programa, el bucle externo **i** se recorre del 2 al 100. El bucle interno prueba con éxito todos los números superiores a 2 para **i**, imprimiendo todos los que se dividen exactamente entre **i**. Un desafío adicional: el programa anterior puede ser más eficiente. ¿Puede observar cómo? (Sugerencia: es posible reducir el número de iteraciones en el bucle interno.)

Comprobación de dominio del módulo 3

1. Escriba un programa que lea caracteres del teclado hasta que se reciba un punto. Haga que el programa cuente el número de espacios y reporte el total al final del programa.
2. Muestre la forma general de la escalera **if-else-if**.
3. Dado:

```
if(x < 10)
  if(y > 100) {
    if(!hecho) x = z;
    else y = z;
```

```
  }
else System.out.println("error");  // ¿qué pasa si?
```

¿a qué **if** se asocia el último **else**?

4. Muestre la instrucción **for** para un bucle que cuenta de 1000 a 0 de –2 en –2.
5. ¿El siguiente fragmento es válido?

```
for(int i = 0; i < num; i++)
  suma += i;

cuenta = i;
```

6. Explique lo que **break** lleva a cabo. Asegúrese de explicar ambas formas.
7. En el siguiente fragmento, después de que se ejecuta la instrucción **break**, ¿qué se despliega?

```
for(i = 0; i < 10; i++)  {
  while(corriendo) {
    if(x<y) break;
     // …
  }
  System.out.println("Después de while");
}
System.out.println("Después de for");
```

8. ¿Qué imprime el siguiente fragmento?

```
for(int i = 0; i<10; i++) {
  System.out.println(i + " ");
  If(i%2) == 0) continue;
  System.out.println();
}
```

9. La expresión de iteración en un bucle **for** no siempre requiere modificar la variable de control del bucle en una cantidad fija. En cambio, la variable puede cambiar de manera arbitraria. Empleando este concepto, escriba un programa que use un bucle **for** para generar y desplegar la progresión 1, 2 , 4, 8, 16, 32, etcétera.
10. Las letras minúsculas en ASCII están separadas de las mayúsculas por 32 caracteres. Por lo tanto, para convertir una minúscula en mayúscula se restan 32. Use esta información para escribir un programa que lea caracteres del teclado. Haga que todas las minúsculas se conviertan en mayúsculas y todas las mayúsculas en minúsculas desplegando el resultado. No le haga cambios a ningún otro carácter. Logre que el programa se detenga cuando el usuario oprima el punto. Al final, haga que el programa despliegue el número de cambios de letras que se realizaron.
11. ¿Qué es un bucle infinito?
12. Cuando utiliza **break** con una etiqueta, ¿la etiqueta debe estar en un bloque que contenga **break**?

Módulo 4

Introducción a clases, objetos y métodos

HABILIDADES FUNDAMENTALES

4.1 Conozca los fundamentos de las clases
4.2 Comprenda cómo se crean los objetos
4.3 Comprenda cómo se asignan las variables de referencia
4.4 Cree métodos, valores devueltos y parámetros de uso
4.5 Use la palabra clave **return**
4.6 Regrese un valor de un método
4.7 Agregue parámetros a un método
4.8 Utilice constructores
4.9 Cree constructores con parámetros
4.10 Comprenda **new**
4.11 Comprenda la recolección de basura y los finalizadores
4.12 Use la palabra clave **this**

Antes de que proceda con su estudio de Java, necesita aprender acerca de las clases. Las clases son la esencia de Java pues representan los cimientos sobre los que todo el lenguaje Java está construido. Así pues, la clase define la naturaleza de un objeto. Como tal, la clase forma la base de la programación orientada a objetos en Java. Dentro de una clase están definidos los datos y el código que actúan sobre tales datos. El código está contenido en métodos. Debido a que en Java son fundamentales, en este módulo se presentarán clases, objetos y métodos. La comprensión básica de estas características le permitirá escribir programas más complejos y entender mejor ciertos elementos clave de Java que se describen en el siguiente módulo.

HABILIDAD FUNDAMENTAL 4.1

Fundamentos de las clases

Desde el inicio de este libro hemos estado empleando clases debido a que toda la actividad en un programa de Java ocurre dentro de una clase. Por supuesto, sólo se han utilizado clases extremadamente simples y aún no se ha aprovechado la mayor parte de sus funciones. Como verá, las clases son sustancialmente más poderosas que las clases limitadas que se han presentado hasta el momento.

Empecemos por revisar los fundamentos: una clase es una plantilla que define la forma de un objeto y especifica los datos y el código que operarán sobre esos datos. Java usa una especificación de clase para construir *objetos*. Los objetos son *instancias* de una clase. Por lo tanto, una clase es, en esencia, un conjunto de planos que especifican cómo construir un objeto. Es importante que el siguiente tema quede claro: una clase es una abstracción lógica. No es sino hasta que se crea un objeto de esa clase que una representación física de dicha clase llega a existir en la memoria.

Otro tema: recuerde que a los métodos y las variables que constituyen una clase se les denomina *miembros* de la clase. Los miembros de datos también son conocidos como *variables de instancia*.

La forma general de una clase

Cuando define una clase, declara su forma y su naturaleza exactas, lo cual lleva a cabo al especificar las variables de instancia que contiene y los métodos que operan sobre estas variables. Aunque es posible que las clases muy simples contengan solamente métodos o variables de instancia, casi todas las clases reales contienen ambos.

Una clase se crea empleando la palabra clave **class**. A continuación se muestra la forma general de una definición de clase:

```
class nombreclase {
  // declare variables de instancia
  tipo var1;
  tipo var2;
  //...
  tipo varN;

  // declare métodos
  tipo método1(parámetros) {
```

```
  // cuerpo del método
  }
  tipo método2(parámetros) {
     // cuerpo del método
  }
     // ...
  tipo métodoN(parámetros) {
     // cuerpo del método
  }
}
```

Aunque no hay reglas sintácticas que así lo dicten, una clase bien diseñada debe definir una y sólo una entidad lógica. Por ejemplo, una clase que almacena nombres y números de teléfono no almacenará información acerca de la bolsa de valores, el promedio de precipitación pluvial, los ciclos de las manchas solares u otra información no relacionada. Lo importante aquí es que una clase bien diseñada agrupa información conectada de manera lógica. Si coloca información no relacionada en la misma clase, ¡desestructurará rápidamente su código!

Hasta este momento, las clases que se han estado empleando sólo tienen un método: **main()**. Pronto verá cómo crear otras clases. Sin embargo, tome en cuenta que la forma general de una clase no específica un método **main()**. Sólo se requiere éste si la clase es el punto de partida de su programa. Además, los applets no requieren un **main()**.

Definición de una clase

Para ilustrar las clases, desarrollaremos una clase que encapsule información acerca de automotores como coches, camionetas y camiones. Esta clase será **Automotor** y almacenará tres elementos de información acerca de un vehículo: el número de pasajeros que puede tener, la capacidad del tanque de combustible y su consumo promedio de gasolina (en kilómetros por litro).

A continuación se presenta la primera versión de **Automotor**. Define tres variables de instancia: **pasajeros**, **tanquegas** y **kpl**. Observe que **Automotor** no contiene métodos, por lo tanto, se trata actualmente de una clase que sólo contiene datos. (En secciones posteriores se le agregarán métodos.)

```
class Automotor {
  int pasajeros;     // número de pasajeros
  int tanquegas;     // capacidad del tanque en litro
  int kpl;           // consumo de gasolina en km por litro
}
```

Una definición de **class** crea un nuevo tipo de datos. En este caso, al nuevo tipo de datos se le llama **Automotor**. Usará este nombre para declarar objetos de tipo **Automotor**. Recuerde que una definición de **class** es sólo una descripción de tipo, por lo que no crea un objeto real. Por consiguiente, el código anterior no hace que ningún objeto de **Automotor** cobre vida.

Para crear realmente un objeto de **Automotor**, debe usar una instrucción como la siguiente:

```
Automotor minivan = new Automotor();  // crea un objeto de Automotor llamado
minivan
```

Después de que esta instrucción se ejecuta, **minivan** será una instancia de **Automotor** y, por lo tanto, tendrá una realidad "física". Por el momento, no se preocupe de los detalles de esta instrucción.

Cada vez que cree la instancia de una clase, estará creando un objeto que contenga su propia copia de cada variable de instancia definida por la clase. En consecuencia, todos los objetos de **Automotor** contendrán sus propias copias de las variables de instancia **pasajeros**, **tanquegas** y **kpl**. Para acceder a estas variables, usará el operador de punto (.). El *operador de punto* vincula el nombre de un objeto con el de un miembro. Ésta es la forma general del operador de punto:

objeto.miembro

El objeto se especifica a la izquierda y el miembro a la derecha. Por ejemplo, para asignar a la variable **tanquegas** de **minivan** el valor 60, use la siguiente instrucción:

```
minivan.tanquegas = 60;
```

En general, puede usar el operador de punto para acceder a variables de instancia y métodos.

He aquí un programa completo que usa la clase **Automotor**:

```
/* Programa que usa la clase Automotor.

   Llame a este archivo AutomotorDemo.java
*/
class Automotor {
  int pasajeros;    // número de pasajeros
  int tanquegas;    // capacidad del tanque en litro
  int kpl;          // consumo de gasolina en km por litro
}

// Esta clase declara un objeto de tipo Automotor.
class AutomotorDemo {
  public static void main(String args[]) {
    Automotor minivan = new Automotor();
    int rango;

    // asigna valores a campos de minivan
    minivan.pasajeros = 7;
    minivan.tanquegas = 60;
    minivan.kpl = 6;

    // calcula el rango suponiendo un tanque lleno de gas
    rango = minivan.tanquegas * minivan.kpl;
```

Observe el uso del operador de punto para acceder a un miembro.

```
    System.out.println("Una minivan puede transportar " + minivan.pasajeros +
                       " pasajeros con un rango de " + rango);
  }
}
```

Debe llamar con el nombre de **AutomotorDemo.java** al archivo que contiene este programa porque el método **main()** está en la clase **AutomotorDemo**, no en la clase **Automotor**. Cuando compile este programa, encontrará que se han creado dos archivos **.class**, uno para **Automotor** y otro para **AutomotorDemo**. El compilador de Java coloca automáticamente cada clase en su propio archivo **.class**. No es necesario que ambas clases estén en el mismo archivo fuente. Puede poner cada clase en sus propios archivos, llamados **Automotor.java** y **AutomotorDemo.java**, respectivamente.

Para ejecutar este programa, debe ejecutar **AutomotorDemo.class**. Se despliega la siguiente salida:

```
Una minivan puede transportar 7 pasajeros con un rango de 360
```

Antes de seguir adelante, revisemos un principio fundamental: cada objeto tiene sus propias copias de las variables de instancia definidas para su clase. Por lo tanto, el contenido de las variables en un objeto puede diferir del contenido de las variables en otro. No existe una conexión entre los dos objetos excepto por el hecho de que ambos objetos son del mismo tipo. Por ejemplo, si tiene dos objetos **Automotor**, cada uno tiene su propia copia de **pasajeros**, **tanquegas** y **kpl**, y el contenido de éstos puede diferir entre los dos objetos. El siguiente programa lo demuestra. (Observe que la clase con **main()** ahora se llama **DosAutomotores**.)

```
// Este programa crea dos objetos de Automotor.

class Automotor {
  int pasajeros;    // número de pasajeros
  int tanquegas;    // capacidad del tanque en litros
  int kpl;          // consumo de gasolina en km por litro
}

// Esta clase declara un objeto de tipo Automotor.
class DosAutomotores {
  public static void main(String args[]) {
    Automotor minivan = new Automotor();
    Automotor carrodepor = new Automotor();

    int rango1, rango2;

    // asigna valores en campos de minivan
    minivan.pasajeros = 7;
    minivan.tanquegas = 60;
    minivan.kpl = 6;
```

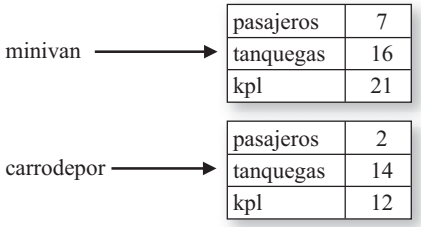
Recuerde que **minivan** y **carrodepor** hacen referencia a objetos separados

```
    // asigna valores en campos de carrodepor
    carrodepor.pasajeros = 2;
    carrodepor.tanquegas = 50;
    carrodepor.kpl = 4;

    // calcula el rango suponiendo un tanque lleno de gas
    rango1 = minivan.tanquegas * minivan.kpl;
    rango2 = carrodepor.tanquegas * carrodepor.kpl;

    System.out.println("Una minivan puede transportar " + minivan.pasajeros +
                       " pasajeros con un rango de " + rango1);
    System.out.println("Un carro deportivo puede transportar " + carrodepor.
pasajeros +
                       " pasajeros con un rango de " + rango2);
  }
}
```

Aquí se muestra la salida producida por este programa:

```
Una minivan puede transportar 7 pasajeros con un rango de 1260
Un carro deportivo puede transportar 2 pasajeros con un rango de 200
```

Como verá, los datos de **minivan** están completamente separados de los contenidos en **carrodepor**. La siguiente ilustración describe esta situación.



Comprobación de avance

1. ¿Cuáles son las dos cosas que una clase contiene?

2. ¿Cuál operador se usa para acceder a los miembros de una clase a través de un objeto?

3. Cada objeto tiene su propia copia de las _____ de la clase.

1. Código y datos. En Java, esto significa métodos y variables de instancia.
2. El operador de punto.
3. Variables de instancia.

HABILIDAD FUNDAMENTAL 4.2

Cómo se crean los objetos

En los programas anteriores se usó la siguiente línea para declarar un objeto de tipo **Automotor**:

```
Automotor minivan = new Automotor();
```

Esta declaración tiene dos funciones. En primer lugar, declara una variable llamada **minivan** del tipo de clase **Automotor**. Esta variable no define un objeto, sino que simplemente es una variable que puede *hacer referencia* a un objeto. En segundo lugar, la declaración crea una copia del objeto y asigna a **minivan** una referencia a dicho objeto. Esto se hace al usar el operador **new**, el cual asigna de forma dinámica (es decir, en tiempo de ejecución) memoria para un objeto y devuelve una referencia a él. Esta referencia es, más o menos, la dirección en memoria del objeto asignado por **new**. Luego, esta referencia se almacena en una variable. Por lo tanto, en Java, todos los objetos de clase deben asignarse dinámicamente.

Los dos pasos combinados de la instrucción anterior pueden reescribirse de esta manera para mostrar cada paso individualmente.

```
Automotor minivan; // declara referencia a un objeto
Minivan = new Automotor();  // asigna un objeto de Automotor
```

La primera línea declara **minivan** como referencia a un objeto de tipo **Automotor**. Por lo tanto, **minivan** es una variable que puede hacer referencia a un objeto, pero no es un objeto en sí. En este sentido, minivan contiene el valor **null**, que significa que no alude a un objeto. La siguiente línea crea un nuevo objeto de Automotor y asigna a minivan una referencia a él. Ahora, **minivan** está vinculado con un objeto.

HABILIDAD FUNDAMENTAL 4.3

Variables de referencia y asignación

En una operación de asignación, las variables de referencia a objetos actúan de manera diferente a las variables de un tipo primitivo, como **int**. Cuando asigna una variable de tipo primitivo a otra, la situación es sencilla: la variable de la izquierda recibe una *copia* del *valor* de la variable de la derecha. Cuando asigna la variable de referencia de un objeto a otro, la situación se vuelve un poco más complicada porque usted cambia el objeto al cual se refiere la variable de referencia. El efecto de esta diferencia puede arrojar algunos resultados contraproducentes. Por ejemplo, considere el siguiente fragmento:

```
Automotor carro1 = new Automotor();
Automotor carro2 = carro1;
```

A primera vista, resulta fácil pensar que **carro1** y **carro2** aluden a objetos diferentes, pero no es así, pues ambos se refieren al *mismo* objeto. La asignación de **carro1** a **carro2** simplemente hace que

carro2 se refiera al mismo objeto que **carro1**. Por lo tanto, **carro1** y **carro2** pueden actuar sobre el objeto. Por ejemplo, después de que se ejecuta la asignación

```
carro1.kpl = 9;
```

estas dos instrucciones **println()**

```
System.out.println(carro1.kpl);
System.out.println(carro2.kpl);
```

despliegan el mismo valor: 9.

Aunque **carro1** y **carro2** hacen referencia al mismo objeto, no están vinculados de ninguna otra manera. Por ejemplo, una asignación posterior a **carro2** simplemente cambia el objeto al que **carro2** hace referencia. Por ejemplo,

```
Automotor carro1 = new Automotor();
Automotor carro2 = carro1;
Automotor carro1 = new Automotor();

carro2 = carro3; // ahora carro2 y carro3 hacen referencia al mismo objeto.
```

Después de que esta secuencia se ejecuta, **carro2** hace referencia al mismo objeto que **carro3**. El objeto al que hace referencia **carro1** permanece sin cambio.

Comprobación de avance

1. Explique lo que ocurre cuando una variable de referencia a un objeto se asigna a otro objeto.
2. Suponiendo una clase llamada **miClase**, muestre cómo se crea un objeto llamado **ob**.

HABILIDAD FUNDAMENTAL 4.4

Métodos

Como ya se explicó, las variables de instancia y los métodos son los elementos de las clases. Hasta ahora, la clase **Automotor** contiene datos, pero no métodos. Aunque las clases que sólo contienen datos son perfectamente válidas, la mayor parte de las clases tiene métodos. Los métodos son subrutinas que manipulan los datos definidos por la clase y, en muchos casos, proporcionan acceso a esos datos. En la mayoría de los casos, otras partes de su programa interactuarán con una clase a través de sus métodos.

1. Cuando una variable de referencia a objeto se asigna a otro objeto, ambas variables hacen referencia al mismo objeto. *No* se hace una copia del objeto.
2. MiClase on = new MiClase( );

Un método contiene una o más instrucciones. En un código de Java bien escrito, cada método sólo realiza una tarea y tiene un nombre. Este nombre es el que se utiliza para llamar al método. En general, puede asignarle a un método el nombre que usted desee. Sin embargo, recuerde que **main()** está reservado para el método que empieza la ejecución de su programa. Además, no debe usar palabras clave de Java para los nombres de métodos.

Al denotar métodos en texto, se ha usado y se seguirá usando en este libro una convención que se ha vuelto común cuando se escribe acerca de Java: un método tendrá un paréntesis después de su nombre. Por ejemplo, si el nombre de un método es **obtenerval**, se escribirá **obtenerval()** cuando su nombre se use en una frase. Esta notación le ayudará a distinguir los nombres de variables de los nombres de métodos en este libro.

La forma general de un método se muestra a continuación:

```
tipo-dev nombre(lista-parámetros) {
  // cuerpo del método
}
```

Aquí, *tipo-dev* especifica el tipo de datos que es devuelto por el método. Éste puede ser cualquier tipo válido, incluyendo tipos de clase que usted puede crear. Si el método no devuelve un valor, su tipo devuelto deberá ser **void**. El nombre del método está especificado por *nombre*. Éste puede ser cualquier identificador legal diferente al que ya usan otros elementos dentro del alcance actual. *Lista-parámetros* es una secuencia de pares tipo-identificador separados por comas. Los parámetros son, en esencia, variables que reciben el valor de los *argumentos* que *pasaron* al método al momento de que éste fue llamado. Si el método no tiene parámetros, la lista de parámetros estará vacía.

Adición de un método a la clase Automotor

Como se acaba de explicar, los métodos de una clase suelen manipular y proporcionar acceso a los datos de la clase. Con esto en mente, recuerde que **main()** en los ejemplos anteriores calculó el rango de un automotor al multiplicar su consumo de gasolina por la capacidad de su tanque. Aunque es técnicamente correcta, ésta no es la mejor manera de manejar este cálculo. La propia clase **Automotor** maneja mejor el cálculo del rango de un automotor. La razón de que esta conclusión sea fácil de comprender es que el rango de un vehículo depende de la capacidad del tanque y la tasa de consumo de gasolina. Ambas cantidades están encapsuladas en **Automotor**. Al agregar a **Automotor** un método que calcule el rango, mejorará su estructura orientada a objetos.

Para agregar un método a **Automotor**, debe especificarlo dentro de una declaración de **Automotor**. Por ejemplo, la siguiente versión de **Automotor** contiene un método llamado **rango()** que despliega el rango del vehículo.

```
// Agrega rango a Automotor.

class Automotor {
  int pasajeros;    // número de pasajeros
  int tanquegas;    // capacidad del tanque en litro
```

```
  int kpl;          // consumo de gasolina en km por litro

  // Despliega el rango.
  void rango() {
    System.out.println("El rango es " + tanquegas * kpl);
  }
}

class AgregarMet {
  public static void main(String args[]) {
    Automotor minivan = new Automotor();
    Automotor carrodepor = new Automotor();

    int rango1, rango2;

    // asigna valores en campos de minivan
    minivan.pasajeros = 7;
    minivan.tanquegas = 60;
    minivan.kpl = 6;

    // asigna valores en campos de carrodepor
    carrodepor.pasajeros = 2;
    carrodepor.tanquegas = 50;
    carrodepor.kpl = 4;


    System.out.print("Una minivan puede transportar " + minivan.pasajeros +
                     ". ");

    minivan.rango(); // despliega rango de minivan

    System.out.print("Un carro deportivo puede transportar " + carrodepor.
pasajeros + ".");

    carrodepor.rango(); // despliega rango de carrodepor.
  }
}
```

El método **rango()** está contenido dentro de la clase **Automotor.**

Observe que **tanquegas** y **klp** se usan de manera directa, sin el operador de punto.

Este programa genera la siguiente salida:

```
Una minivan puede transportar 7 pasajeros con un rango de 1260
Un carro deportivo puede transportar 2 pasajeros con un rango de 200
```

Revisemos los elementos clave de este programa empezando por el propio método **rango()**. La primera línea de **rango()** es

```
void rango() {
```

La línea declara un método llamado **rango** que no tiene parámetros. El tipo que devuelve es **void**; así que **rango( )** no devuelve un valor al método que lo llama. La línea termina con la llave de apertura del método del cuerpo.

El cuerpo de **rango( )** es la siguiente línea única:

```
System.out.println("El rango es " + tanquegas * kpl);
```

La instrucción despliega el rango del vehículo al multiplicar **tanquegas** por **kpl**. Debido a que cada objeto de tipo **Automotor** tiene su propia copia de **tanquegas** y **kpl**, cuando se llama a **rango( )**, el cálculo del rango usa copias de esas variables en el objeto que llama.

El método **rango( )** termina cuando se encuentra su llave de cierre. Esto hace que el control del programa se transfiera de nuevo al método que hace la llamada.

Ahora, revise esta línea de código a partir del interior de **main( )**:

```
minivan.rango();
```

Esta instrucción invoca el método **rango( )** de **minivan**, es decir, llama a **rango( )** relacionado con el objeto minivan empleando el nombre del objeto seguido del operador de punto. Cuando se llama a un método, el control del programa se transfiere al método. Cuando el método termina, el control se regresa al que llama, y la ejecución se reanuda con la línea de código que sigue a la llamada.

En este caso, la llamada a **minivan.rango( )** despliega el rango del vehículo definido por **minivan**. De manera similar, la llamada a **carrodepor.rango( )** despliega el rango del vehículo definido por **carrodepor**. Cada vez que **rango( )** es invocado, éste despliega el rango del objeto especificado.

Debe tomarse en cuenta un elemento muy importante dentro del método **rango( )**: hay referencias directas a las variables de instancia **tanquegas** y **kpl**; no es necesario antecederlas con un nombre de objeto o un operador de punto. Cuando un método usa una variable de instancia que está definida por su clase, lo hace directamente, sin la referencia explícita a un objeto y sin el uso del operador de punto. Si lo reflexiona, resulta fácil comprenderlo. Un método siempre se invoca en relación con algún objeto de su clase. Una vez que esta invocación ha ocurrido, se conoce al objeto. Por lo tanto, dentro de un método no es necesario especificar el objeto por segunda ocasión. Esto significa que **tanquegas** y **kpl** dentro de **rango( )** hacen referencia implícita a las copias de estas variables encontradas en el objeto que invoca a **rango( )**.

HABILIDAD FUNDAMENTAL 4.5

Regreso de un método

En general, hay dos condiciones que causan el regreso de un método: la primera, como se muestra en el método **rango( )**, es cuando se encuentra la llave de cierre del método; y la segunda es cuando se ejecuta una instrucción **return**. Hay dos formas de **return**: para usarla en métodos **void** (los que no devuelven un valor) y para valores de regreso. La primera forma se examina a continuación. En la siguiente sección se explicará cómo devolver valores.

En un método **void**, puede usar la terminación inmediata de un método empleando esta forma de **return**:

return;

Cuando se ejecuta esta instrucción, el control del programa regresa al método que llama, omitiendo cualquier código restante del método. Por ejemplo, revise este método:

```
void miMet() {
  int i;

  for(i=0; i<10; i++) {
    if(i == 5) return; // se detiene en 5
    System.out.println();
  }
}
```

Aquí, el bucle **for** sólo se ejecutará de 0 a 5 porque una vez que **i** es igual a 5, el método regresa.

Se permite tener varias instrucciones de regreso en un método, sobre todo cuando hay dos o más rutas fuera de él. Por ejemplo:

```
void miMet() {
  // ...
  if(hecho) return;
  // ...
  if(error) return;
}
```

Aquí, el método regresa si se cumple o si un error ocurre. Sin embargo, tenga cuidado, porque la existencia de muchos puntos de salida de un método puede destruir su código; así que evite su uso de manera casual. Un método bien diseñado se caracteriza por puntos de salida bien definidos.

En resumen: un método **void** puede regresar mediante una de las siguientes formas: ya sea que se llegue a su llave de cierre o que se ejecute una instrucción **return**.

HABILIDAD FUNDAMENTAL 4.6

Devolución de un valor

Aunque los métodos con un tipo de devolución **void** no son raros, casi todos los métodos devolverán un valor. En realidad, la capacidad de regresar un valor es una de las características más útiles de un método. Un ejemplo de un valor devuelto es cuando usamos la función **sqrt()** para obtener una raíz cuadrada.

Los valores devueltos se usan para diversos fines en programación. En algunos casos, como con **sqrt()**, el valor devuelto contiene el resultado de algunos cálculos. En otros casos, el valor devuelto puede indicar simplemente éxito o fracaso. En otros más, puede contener un código de estatus. Cualquiera que sea el propósito, el uso de valores de retorno de método constituye una parte integral de la programación en Java.

Los métodos regresan un valor a la rutina que llaman empleando esta forma de **return**:

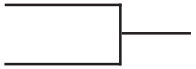
return *valor*;

Aquí, *valor* es el valor devuelto.

Puede usar un valor devuelto para mejorar la implementación de **rango()**. En lugar de desplegar el rango, un método mejor consiste en hacer que **rango()** calcule el rango y devuelva este valor. Una de las ventajas de este método es que puede usar el valor para otros cálculos. El siguiente ejemplo modifica **rango()** para que devuelva el rango en lugar de desplegarlo.

```
// Uso de un valor devuelto.

class Automotor {
  int pasajeros;    // número de pasajeros
  int tanquegas;    // capacidad del tanque en litro
  int kpl;          // consumo de gasolina en km por litro

  // Devuelve el rango.
  int rango() {
    return kpl * tanquegas;  <-- Devuelve el rango dado para un automotor determinado.
  }
}

class RetMet {
  public static void main(String args[]) {
    Automotor minivan = new Automotor();
    Automotor carrodepor = new Automotor();

    int rango1, rango2;

    // asigna valores en campos de minivan
    minivan.pasajeros = 7;
    minivan.tanquegas = 60;
    minivan.kpl = 6;

    // asigna valores en campos de carrodepor
    carrodepor.pasajeros = 2;
    carrodepor.tanquegas = 50;
    carrodepor.kpl = 4;
```

```
    // obtiene los rangos
    rango1 = minivan.rango();
    rango2 = carrodepor.rango();
```




```
    System.out.println("Una minivan puede transportar " + minivan.pasajeros +
                       " pasajeros con rango de " + rango1 + " km");

    System.out.println("Un carro deportivo puede transportar " + carrodepor.
pasajeros + " pasajeros con rango de " + rango2 + " km");

  }
}
```

Ésta es la salida:

```
Una minivan puede transportar 7 pasajeros con rango de 1260 km
Un carro deportivo puede transportar 2 pasajeros con un rango de 200 km
```

En el programa, observe que cuando se llama a **rango()**, éste se pone a la derecha de la instrucción de asignación. A la izquierda está una variable que recibirá el valor devuelto por **rango()**. Por lo tanto, después de que se ejecuta

```
rango1 = minivan.rango();
```

el rango del objeto **minivan** se almacena en **rango1**.

Observe que **rango()** ahora tiene un tipo de retorno **int**. Esto significa que devolverá un valor entero. El tipo devuelto por un método es importante porque el tipo de datos debe ser compatible con el tipo devuelto que el método especifica. Así que, si quiere que un método devuelva datos de tipo **double**, su tipo devuelto debe ser **double**.

Aunque el programa anterior es correcto, no está escrito con la eficiencia que podría tener. De manera específica, no son necesarias las variables **rango1** o **rango2**. Es posible usar directamente una llamada a **rango()** en la instrucción **println()** como se muestra aquí:

```
System.out.println("Una minivan puede transportar " + minivan.pasajeros +
                   " pasajeros con rango de " + minivan.rango() + " km");
```

En este caso, cuando se ejecuta **println()**, se llama automáticamente a **minivan.rango()** y su valor se pasará a **println()**. Más aún, puede usar una llamada a **rango()** cada vez que se requiera el rango de un objeto de **Automotor**. Por ejemplo, esta instrucción compara los rangos de los dos vehículos:

```
if(a1.rango() > a2.rango() System.out.println("a1 tiene un rango mayor");
```

HABILIDAD FUNDAMENTAL 4.7

Uso de parámetros

Es posible pasar uno o más valores a un método cuando éste es llamado. Como se explicó, un valor que pasa a un método es un *argumento*. Dentro del método, la variable que recibe el argumento es un *parámetro*. Los parámetros se declaran dentro del paréntesis que sigue al nombre del método. La sintaxis de declaración del parámetro es la misma que la que se usa para las variables. Un parámetro se encuentra dentro del alcance de su método y, además de las tareas especiales de recibir un argumento, actúa como cualquier otra variable local.

He aquí un ejemplo simple en el que se utiliza un parámetro. Dentro de la clase **RevNum**, el método **esPar()** devuelve **true** si el valor que se pasa es par. De lo contrario, devuelve **false**. Por lo tanto, **esPar()** tiene un tipo de regreso **boolean**.

```
// Un ejemplo simple en el que se utiliza un parámetro.

class RevNum {
  // devuelve true si x es par
  boolean esPar(int x) {
    if((x%2) == 0) return true;
    else return false;
  }
}

class ParamDemo {
  public static void main(String args[]) {
    RevNum e = new RevNum();

    if(e.esPar(10)) System.out.println("10 es par.");

    if(e.esPar(9)) System.out.println("9 es par.");

    if(e.esPar(8)) System.out.println("8 es par.");

  }
}
```

Aquí, **x** es un parámetro entero de **esPar()**.

Pasa argumentos a **esPar()**.

He aquí la salida producida por el programa:

```
10 es par
8 es par
```

En el programa, se llama tres veces a **esPar()**, y cada vez que esto sucede se pasa un valor diferente. Echemos un vistazo de cerca al proceso. En primer lugar, observe cómo se llama a **esPar()**:

el argumento se especifica entre los paréntesis y cuando se llama a **esPar()** por primera vez, se pasa el valor 10. De ahí que, cuando se empieza a ejecutar **esPar()**, el parámetro **x** recibe el valor 10. En la segunda llamada, el argumento es 9, y **x** entonces tiene el valor 9. En la tercera llamada, el argumento es 8, que es el valor que recibe **x**. Lo que debe notar es que el valor que se pasa como argumento cuando se llama a **esPar()** es el valor recibido por su parámetro, en este caso, **x**.

Un método puede tener más de un parámetro. Simplemente declare cada parámetro separándolos mediante una coma. Por ejemplo, la clase **Factor** define un método llamado **esFactor()** que determina si el primer parámetro es factor del segundo.

```
class Factor {
  boolean esFactor(int a, int b) {
    if( (b % a) == 0) return true;
    else return false;
  }
}
class esFact {
  public static void main(String args[]) {
    Factor x = new Factor();

    if(x.esFactor(2, 20)) System.out.println("2 es factor");
    if(x.esFactor(3, 20)) System.out.println("esto no se muestra");

  }
}
```

Este método tiene 2 parámetros.

Pasa dos argumentos a **esFactor()**.

Observe que cuando se llama a **esFactor()**, los argumentos también están separados por comas.

Cuando se usan varios parámetros, cada uno especifica su propio tipo, el cual puede diferir de los demás. Por ejemplo, lo siguiente es perfectamente válido:

```
int MiMet(int a, double b, float c) {
 // ...
```

Adición de un método con parámetros a un automotor

Puede usar un método con parámetros para agregar una nueva función a la clase **Automotor**: la capacidad de calcular la cantidad de gasolina necesaria para recorrer una distancia determinada. A este nuevo método se le llamará **gasnecesaria()**. Este método toma el número de kilómetros que quiere manejar y devuelve el número de litros requeridos de gasolina. El método **gasnecesaria()** se define así:

```
doble gasnecesaria (int miles) {
  return (double) miles / kpl
}
```

Tome en cuenta que este método regresa un valor de tipo **double**. Esto es útil ya que la cantidad de gasolina necesaria para una distancia tal vez no sea un número constante.

La clase **Automotor** completa que incluye tanquegas se muestra a continuación

```
/*
   Agrega un método con parámetros que calcula la
   gasolina necesaria para una distancia determinada.
*/

class Automotor {
  int pasajeros;    // número de pasajeros
  int tanquegas;    // capacidad del tanque en litro
  int kpl;          // consumo de gasolina en km por litro

  // Devuelve el rango.
  int rango() {
    return kpl * tanquegas;
  }

  // calcula la gasolina necesaria para una distancia.
  double gasnecesaria(int kilómetros) {
    return (double) miles / kpl;
  }
}

class CalcGas {
  public static void main(String args[]) {
    Automotor minivan = new Automotor();
    Automotor carrodepor = new Automotor();
    double litros;
    int dist = 252;

    // asigna valores en campos de minivan
    minivan.pasajeros = 7;
    minivan.tanquegas = 60;
    minivan.kpl = 6;

    // asigna valores en campos de carrodepor
    carrodepor.pasajeros = 2;
    carrodepor.tanquegas = 50;
    carrodepor.kpl = 4;
```

```
    litros = minivan.gasnecesaria(dist);

    System.out.println("Para recorrer " + dist + " km una minivan necesita " +
                       litros + " litros de gasolina.");

    litros = carrodepor.gasnecesaria(dist);

    System.out.println("Para recorrer " + dist + " km un carro deportivo
necesita " + litros + " litros de gasolina.");

  }
}
```

Ésta es la salida del programa:

```
Para recorrer 252 km una minivan necesita 42.0 litros de gasolina.
Para recorrer 252 km un carro deportivo necesita 63.0 litros de gasolina.
```

Comprobación de avance

1. ¿Cuándo se debe acceder a una variable de instancia o a un método mediante la referencia a un objeto empleando el operador de punto? ¿Cuándo puede usarse directamente una variable o un método?
2. Explique la diferencia entre un argumento y un parámetro.
3. Explique las dos maneras en las que un método puede regresar a su llamada.

1. Cuando se accede a una variable de instancia mediante un código que no es parte de la clase en la que está definida la variable de instancia, debe realizarse mediante un objeto, con el uso de un operador de punto. Sin embargo, cuando se accede a una variable de instancia con un código que es parte de la misma clase que la variable de instancia, es posible hacer referencia directa a dicha variable. Lo mismo se aplica a los métodos.
2. Un *argumento* es un valor que se pasa a un método cuando es invocado. Un *parámetro* es una variable definida por un método que recibe el valor de un argumento.
3. Es posible hacer que un método regrese con el uso de la instrucción **return**. Si el método cuenta con un tipo de regreso **void**, entonces el método también regresará cuando se llegue a sus llaves de cierre. Los métodos que no son **void** deben devolver un valor, de modo que regresar al alcanzar la llave de cierre no es una opción.

Proyecto 4.1 Creación de una clase Ayuda

`ClaseAyudaDemo.java`

Si se intentara resumir en una frase la esencia de una clase, éste sería el resultado: una clase encapsula la funcionalidad. Por supuesto, el truco consiste en saber dónde termina una "funcionalidad" y dónde empieza otra. Como regla general, seguramente usted desea que sus clases sean los bloques de construcción de una aplicación más grande. Para ello, cada clase debe representar una sola unidad funcional que realice acciones claramente delineadas; así que usted querrá que sus clases sean lo más pequeñas posibles, ¡pero no demasiado pequeñas! Es decir, las clases que contienen una funcionalidad extraña confunden y destruyen al código, pero las clases que contienen muy poca funcionalidad están fragmentadas. ¿Cuál es el justo medio? Éste es el punto en el que la *ciencia* de la programación se vuelve el *arte* de la programación. Por fortuna, la mayoría de los programadores se dan cuenta de que ese acto de equilibrio se vuelve más fácil con la experiencia.

Para empezar a obtener esa experiencia, convierta el sistema de ayuda del proyecto 3.3 del módulo anterior en una clase Ayuda. Examinemos el porqué ésta es una buena idea. En primer lugar, el sistema de ayuda define una unidad lógica. Simplemente despliega la sintaxis de las instrucciones de control de Java. Por consiguiente, su funcionalidad es compacta y está bien definida. En segundo lugar, colocar la ayuda en una clase es un método estéticamente agradable. Cada vez que quiera ofrecer el sistema de ayuda a un usuario, simplemente iniciará un objeto de sistema de ayuda. Por último, debido a que la ayuda está encapsulada, ésta puede actualizarse o cambiarse sin que ello ocasione efectos secundarios indeseables en los programas que la utilizan.

Paso a paso

1. Cree un nuevo archivo llamado **ClaseAyudaDemo.java**. Si desea guardar lo que ya tiene escrito, copie el archivo del proyecto 3.3, llamado **Ayuda3.java**, en **ClaseAyudaDemo.java**.
2. Para convertir el sistema de ayuda en una clase, primero debe determinar con precisión lo que constituye el sistema de ayuda. Por ejemplo, en **Ayuda3.java** hay un código para desplegar un menú, ingresar opciones del usuario, revisar respuestas válidas y desplegar información acerca del elemento seleccionado. El programa también se recorre en bucle hasta que se oprime la letra q. Es evidente entonces que el menú, la revisión de una respuesta válida y el despliegue de la información integran al sistema de ayuda, y no la manera en que se obtiene la entrada del usuario y si deben o no procesarse solicitudes repetidas. Por lo tanto, creará una clase que despliegue la información de ayuda y el menú de ayuda, y que compruebe que se lleve a cabo una selección válida. Sus métodos se denominarán **ayudactiva( )**, **mostrarmenu( )** y **escorrecta( )**, respectivamente.

(continúa)

3. Cree el método **ayudactiva()** como se muestra aquí:

```
void ayudactiva(int que) {
    switch(que) {
      case '1':
        System.out.println("if:\n");
        System.out.println("if(condición) instrucción;");
        System.out.println("instrucción else;");
        break;
      case '2':
        System.out.println("switch:\n");
        System.out.println("switch(expresión) {");
        System.out.println("  constante case:");
        System.out.println("    secuencia de instrucciones");
        System.out.println("    break;");
        System.out.println("  // ...");
        System.out.println("}");
        break;
      case '3':
        System.out.println("for:\n");
        System.out.print("for(inic; condición; iteración)");
        System.out.println(" instrucción;");
        break;
      case '4':
        System.out.println("while:\n");
        System.out.println("while(condición) instrucción;");
        break;
      case '5':
        System.out.println("do-while:\n");
        System.out.println("do {");
        System.out.println("  instrucción;");
        System.out.println("} while (condición);");
        break;
      case '6':
        System.out.println("break:\n");
        System.out.println("break; o break etiqueta;");
        break;
      case '7':
        System.out.println("continue:\n");
        System.out.println("continue; o continue etiqueta;");
        break;
  }
  System.out.println();
}
```

4. A continuación, cree el método **mostrarmenu()**:

```
void mostrarmenu() {
  System.out.println("Ayuda habilitada:");
```

```
  System.out.println("  1. if");
  System.out.println("  2. switch");
  System.out.println("  3. for");
  System.out.println("  4. while");
  System.out.println("  5. do-while");
  System.out.println("  6. break");
  System.out.println("  7. continue\n");
  System.out.print("Elija una (q para salir): ");
}
```

5. Cree el método **escorrecta()** que se muestra aquí:

```
boolean escorrecta(int ch) {
  if(ch < '1' | ch > '7' & ch != 'q') return false;
  else return true;
}
```

6. Ensamble los métodos anteriores en la clase **Ayuda**, mostrada aquí:

```
class Ayuda {
  void ayudactiva(int que) {
    switch(que) {
      case '1':
        System.out.println("if:\n");
        System.out.println("if(condición) instrucción;");
        System.out.println("instrucción else;");
        break;
      case '2':
        System.out.println("switch:\n");
        System.out.println("switch(expresión) {");
        System.out.println("  constante case:");
        System.out.println("    secuencia de instrucciones");
        System.out.println("    break;");
        System.out.println("  // ...");
        System.out.println("}");
        break;
      case '3':
        System.out.println("for:\n");
        System.out.print("for(inic; condición; iteración)");
        System.out.println(" instrucción;");
        break;
      case '4':
        System.out.println("while:\n");
        System.out.println("while(condición) instrucción;");
        break;
      case '5':
        System.out.println("do-while:\n");
```

(continúa)

```
        System.out.println("do {");
        System.out.println("  instrucción;");
        System.out.println("} while (condición);");
        break;
      case '6':
        System.out.println("break:\n");
        System.out.println("break; o break etiqueta;");
        break;
      case '7':
        System.out.println("continue:\n");
        System.out.println("continue; o continue etiqueta;");
        break;
    }
    System.out.println();
  }

  void mostrarmenú() {
    System.out.println("Ayuda habilitada:");
    System.out.println("  1. if");
    System.out.println("  2. switch");
    System.out.println("  3. for");
    System.out.println("  4. while");
    System.out.println("  5. do-while");
    System.out.println("  6. break");
    System.out.println("  7. continue\n");
    System.out.print("Elija una (q para salir): ");
  }

  boolean escorrecta(int ch) {
    if(ch < '1' | ch > '7' & ch != 'q') return false;
    else return true;
  }

}
```

7. Por último, reescriba el método **main()** del proyecto 3.3, de modo que use la nueva clase **Ayuda**. Llame a esta clase **ClaseAyudaDemo.java**. A continuación se muestra el listado completo:

```
/*
   Proyecto 4.1

   Convierte el sistema de ayuda del proyecto 3.3
   en una clase Ayuda.
*/

class Ayuda {
```

```
  void ayudactiva(int que) {
    switch(que) {
      case '1':
        System.out.println("if:\n");
        System.out.println("if(condición) instrucción;");
        System.out.println("instrucción else;");
        break;
      case '2':
        System.out.println("switch:\n");
        System.out.println("switch(expresión) {");
        System.out.println("  constante case:");
        System.out.println("    secuencia de instrucciones");
        System.out.println("    break;");
        System.out.println("  // ...");
        System.out.println("}");
        break;
      case '3':
        System.out.println("for:\n");
        System.out.print("for(inic; condición; iteración)");
        System.out.println(" instrucción;");
        break;
      case '4':
        System.out.println("while:\n");
        System.out.println("while(condición) instrucción;");
        break;
      case '5':
        System.out.println("do-while:\n");
        System.out.println("do {");
        System.out.println("  instrucción;");
        System.out.println("} while (condición);");
        break;
      case '6':
        System.out.println("break:\n");
        System.out.println("break; o break etiqueta;");
        break;
      case '7':
        System.out.println("continue:\n");
        System.out.println("continue; o continue etiqueta;");
        break;
  }
  System.out.println();
}

  void mostrarmenú() {
    System.out.println("Ayuda habilitada:");
```

(continúa)

```
    System.out.println("  1. if");
    System.out.println("  2. switch");
    System.out.println("  3. for");
    System.out.println("  4. while");
    System.out.println("  5. do-while");
    System.out.println("  6. break");
    System.out.println("  7. continue\n");
    System.out.print("Elija una (q para salir): ");
  }

  boolean escorrecta(int ch) {
    if(ch < '1' | ch > '7' & ch != 'q') return false;
    else return true;
  }

}

class ClaseAyudaDemo {
  public static void main(String args[])
    throws java.io.IOException {
    char inciso;
    Ayuda objayuda = new Ayuda();

    for(;;) {
      do {
        objayuda.mostrarmenú();
        do {
          inciso = (char) System.in.read();
        } while(inciso == '\n' | inciso == '\r');

      } while( !objayuda.escorrecta(inciso) );

      if(inciso == 'q') break;

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


      objayuda.ayudactiva(inciso);
    }
  }
}
```

Al probar el programa encontrará que funciona igual que antes. La ventaja de este método es que ahora tiene un componente de sistema de ayuda que puede reutilizarse cuando sea necesario.

HABILIDAD FUNDAMENTAL 4.8

Constructores

En los ejemplos anteriores, las variables de instancia de cada objeto **Automotor** tienen que establecerse manualmente mediante una secuencia de instrucciones como:

```
minivan.pasajeros = 7;
minivan.tanquegas = 60;
minivan.kpl = 6;
```

Un método como éste nunca se usaría en un código escrito profesionalmente en Java. Aparte de ser propenso a error (podría olvidar establecer alguno de los campos), existe una mejor manera de realizar esta tarea: el constructor.

Un *constructor* inicializa un objeto cuando este último se crea. Tiene el mismo nombre que su clase y es sintácticamente similar a un método. Sin embargo, los constructores no tienen un tipo de regreso explícito. Por lo general, usará un constructor para dar valores iniciales a las variables de instancia definidas por la clase, o para realizar cualquier otro procedimiento de inicio que se requiera para crear un objeto completamente formado.

Todas las clases tienen constructores, ya sea que los defina o no, porque Java proporciona automáticamente un constructor predeterminado que inicializa todas las variables de miembros en cero. Sin embargo, una vez que haya definido su constructor, el constructor predeterminado ya no se utilizará.

He aquí un ejemplo simple en el que se utiliza un constructor.

```
// Un constructor simple.

class MiClase {
  int x;

  MiClase() {
    x = 10;
  }
}

class ConsDemo {
  public static void main(String args[]) {
    MiClase t1 = new MiClase();
    MiClase t2 = new MiClase();

    System.out.println(t1.x + " " + t2.x);
  }
}
```

Éste es el constructor de **MiClase**.

En este ejemplo, el constructor de **MiClase** es

```
MiClase() {
  x = 10;
}
```

Este constructor asigna el valor 10 a la variable de instancia **x** de **MiClase**. Este constructor es llamado por **new** cuando un objeto se crea. Por ejemplo, en la línea

```
MiClase t1 = new MiClase();
```

se llama al constructor **MiClase()** en el objeto **t1**, dando a **t1.x** el valor 10. Lo mismo aplica para **t2**. Después de la construcción, **t2.x** tiene el valor de 10. Por lo tanto, la salida del programa es

```
10 10
```

HABILIDAD FUNDAMENTAL 4.9

Constructores con parámetros

En el ejemplo anterior se empleó un constructor sin parámetros. Aunque esto resulta adecuado para algunas situaciones, lo más común será que necesite un constructor que acepte uno o más parámetros. Los parámetros se agregan a un constructor de la misma manera en la que se agregan a un método: sólo se les declara dentro del paréntesis después del nombre del constructor. Por ejemplo, en el siguiente caso, a **MiClase** se le ha dado un constructor con parámetros:

```
// Un constructor con parámetros.

class MiClase {
  int x;

  MiClase(int i) {  <-- El constructor tiene un parámetro.
    x = i;
  }
}

class ConsParamDemo {
  public static void main(String args[]) {
    MiClase t1 = new MiClase(10);
    MiClase t2 = new MiClase(88);

    System.out.println(t1.x + " " + t2.x);
  }
}
```

Ésta es la salida de este programa:

```
10 88
```

En esta versión del programa, el constructor **MiClase()** define un parámetro llamado **i**, el cual se usa para inicializar la variable de instancia **x**. Por lo tanto, cuando se ejecuta la línea

```
MiClase t1 = new MiClase(10);
```

se pasa el valor 10 a **i**, que luego se asigna a **x**.

Adición de un constructor a la clase Automotor

Podemos mejorar la clase **Automotor** agregando un constructor que inicialice automáticamente los campos pasajeros, **tanquegas** y **kpl** cuando se construye un objeto. Ponga especial atención en la manera en que se crean los objetos de **Automotor**.

```
// Agregar un constructor.

class Automotor {
  int pasajeros;    // número de pasajeros
  int tanquegas;    // capacidad del tanque en litro
  int kpl;          // consumo de gasolina en km por litro

  // Este es un constructor para Automotor.
  Automotor(int p, int f, int m) {
    pasajeros = p;
    tanquegas = f;
    kpl = m;
  }

  // Regresa el rango.
  int rango() {
    return kpl * tanquegas;
  }

  // calcula la gasolina necesaria para una distancia.
  double gasnecesaria(int km) {
    return (double) km / kpl;
  }
}

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

    // construye automotores completos
    Automotor minivan = new Automotor(7, 60, 6);
    Automotor carrodepor = new Automotor(2, 50, 3);
    double litros;
    int dist = 252;

    litros = minivan.gasnecesaria(dist);

    System.out.println("Para recorrer " + dist + " kms una minivan necesita " +
                       litros + " litros de gasolina.");
```

Constructor de **Automotor.**

```
    litros = carrodepor.gasnecesaria(dist);

    System.out.println("Para recorrer " + dist + " kms un carro deportivo
                       necesita " + litros + " litros de gasolina.");

  }
}
```

Cuando se crean, tanto **minivan** como **carrodepor** se inicializan con el constructor **Automotor()**. Cada objeto es inicializado como se especifica en los parámetros de su constructor. Por ejemplo, en la siguiente línea:

```
Automotor minivan = new Automotor(7, 60, 6);
```

Los valores 7, 60 y 6 se pasan al constructor **Automotor()** cuando **new** crea el objeto. Por lo tanto, las copias de **minivan** de **pasajeros**, **tanquegas** y **kpl** contendrán los valores 7, 60 y 6, respectivamente. La salida de este programa es la misma que en la versión anterior.

Comprobación de avance

1. ¿Qué es un constructor y cuándo se ejecuta?
2. ¿Un constructor tiene un tipo de regreso?

HABILIDAD FUNDAMENTAL 4.10

Nueva visita al operador new

Ahora que sabe más de las clases y sus constructores, echemos un vistazo al operador **new**. Este operador tiene esta forma general:

var-clase = new *nombre-clase*();

En este caso, *var-clase* es una variable del tipo de clase que se está creando. El *nombre-clase* es el nombre de la clase de la que se está creando una instancia. El nombre de la clase seguida de un paréntesis especifica el constructor de la clase. Si una clase no define su propio constructor, **new** usará el constructor predeterminado que Java proporciona. De manera que **new** puede usarse para crear un objeto de cualquier tipo de clase.

1. Un constructor es un método que se ejecuta cuando el objeto de una clase se inicializa. Un constructor se usa para inicializar el objeto que se está creando.
2. No.

Pregunte al experto

P: **¿Porqué no necesito usar** new **para variables de tipos primitivos, como** int **o** float**?**

R: Los tipos primitivos de Java no están implementados como objetos sino que, debido a razones de eficiencia, están implementados como variables "normales". Una variable de tipo primitivo contiene en realidad el valor que usted le ha proporcionado. Como se explicó, las variables de objeto son referencias al objeto. Esta capa de direccionamiento (así como otras funciones del objeto) agregan una carga extra a un objeto que se evita en un tipo primitivo.

Debido a que la memoria es finita, tal vez **new** no pueda asignar memoria a un objeto por falta de memoria suficiente. Si esto sucede, ocurrirá una excepción en tiempo de ejecución. (Aprenderá a manejar ésta y otras excepciones en el módulo 9.) En el caso de los programas que vienen de ejemplo en este libro, no tendrá que preocuparse por quedarse sin memoria, pero debe considerar esta posibilidad en los programas reales que escriba.

HABILIDAD FUNDAMENTAL 4.11

Recolección de basura y finalizadores

Como se ha visto, los objetos se asignan, empleando el operador **new**, de manera dinámica a partir de un almacén de memoria libre. Como se explicó, la memoria no es infinita por lo que la memoria libre puede agotarse. Por lo tanto, es posible que **new** falle porque hay insuficiente memoria libre para crear el objeto deseado. Por tal motivo, un componente clave de cualquier esquema de asignación dinámica es la recuperación de memoria libre a partir de objetos no empleados, lo que deja memoria disponible para una reasignación posterior. En muchos lenguajes de programación, la liberación de la memoria asignada previamente se maneja de manera manual. Por ejemplo, en C++ usted usa el operador **delete** para liberar la memoria que fue asignada. Sin embargo, Java usa un método diferente, más libre de problemas: la *recolección de basura*.

El sistema de recolección de basura de Java reclama objetos automáticamente (lo cual ocurre de manera transparente, tras bambalinas, sin intervención del programador). Funciona así: cuando no existen referencias a un objeto, se supone que dicho objeto ya no es necesario por lo que se libera la memoria ocupada por el objeto. Esta memoria reciclada puede usarse entonces para asignaciones posteriores.

La recolección de basura sólo ocurre de manera esporádica durante la ejecución de su programa. No ocurrirá por el solo hecho de que existan uno o más objetos que ya no se usen. Por razones de eficiencia, la recolección de basura por lo general sólo se ejecutará cuando se cumplan dos condiciones: cuando haya objetos que reciclar y cuando sea necesario reciclarlos. Recuerde que la recolección de basura ocupa tiempo, de modo que el sistema en tiempo de ejecución de Java sólo la lleva a cabo cuando es necesaria. Por consiguiente, no es posible saber con precisión en qué momento tendrá lugar la recolección de basura.

El método finalize()

Es posible definir un método que sea llamado antes de que la recolección de basura se encargue de la destrucción final de un objeto. A este método se le llama **finalize()** y se utiliza con el fin de asegurar que un objeto terminará limpiamente. Por ejemplo, podría usar **finalize()** para asegurarse de que se cierre un archivo abierto que es propiedad de un determinado objeto.

Para agregar un finalizador a una clase, simplemente debe definir el método **finalize()**. En tiempo de ejecución, Java llama a ese método cada vez que está a punto de reciclar un objeto de esa clase. Dentro del método **finalize()** usted especificará las acciones que deben realizarse antes de que un objeto se destruya.

El método **finalize()** tiene la siguiente forma general:

```
protected void finalize()
{
  //aquí va el código de finalización
}
```

En este caso, la palabra clave **protected** es un especificador que evita que un código definido fuera de su clase acceda a **finalize()**. Éste y los demás especificadores de acceso se explicarán en el módulo 6.

Resulta importante comprender que se llama a **finalize()** justo antes de la recolección de basura. No se le llama cuando, por ejemplo, un objeto sale del alcance. Esto significa que usted no puede saber cuando (o incluso si) **finalize()** se ejecutará. Por ejemplo, si su programa termina antes de que la recolección de basura ocurra, **finalize()** no se ejecutará. Por lo tanto, éste debe usarse como procedimiento de "respaldo" para asegurar el manejo apropiado de algún recurso, o bien, para aplicaciones de uso especial, y no como el medio que su programa emplea para su operación normal.

Pregunte al experto

P: **Sé que C++ define a los llamados** destructores**, los cuales se ejecutan automáticamente cuando se destruye un objeto. ¿**Finalize( ) **es parecido a un destructor?**

R: Java no tiene destructores. Aunque es verdad que el método **finalize()** se aproxima a la función de un destructor, no son lo mismo. Por ejemplo, siempre se llama a un destructor de C++ antes de que un objeto salga del alcance; sin embargo, no es posible saber en qué momento se llamará a **finalize()** para un objeto específico. Francamente, como Java emplea la recolección de basura, un destructor no resulta muy necesario.

Proyecto 4.2 Demostración de la finalización

Finalize.java

Debido a que la recolección de basura se aplica de manera esporádica y secundaria, no resultará trivial la demostración del método **finalize()**. Recuerde que se llama a **finalize()** cuando un objeto está por ser reciclado. Como ya se explicó, los objetos no necesariamente se reciclan en cuanto dejan de ser necesarios. El recolector de basura espera hasta que pueda realizar eficientemente su recolección que, por lo general, es cuando hay muchos objetos sin usar. En este sentido, para demostrar el método **finalize()**, a menudo necesitará crear y destruir una gran cantidad de objetos. Y eso es precisamente lo que llevará a cabo en este proyecto.

Paso a paso

1. Cree un nuevo archivo llamado **Finalize.java.**
2. Cree la clase **FDemo** que se muestra aquí:

```
class FDemo {
  int x;

  FDemo(int i) {
    x = i;
  }

  // llamado cuando se recicla el objeto
  protected void finalize() {
    System.out.println("Finaliza " + x);
  }

  // genera un ob que se destruye de inmediato
  void generator(int i) {
    FDemo o = new FDemo(i);
  }

}
```

El constructor asigna a la variable de instancia **x** un valor conocido. En este ejemplo, **x** se usa como ID de un objeto. El método **finalize()** despliega el valor de **x** cuando se recicla un objeto. De especial interés resulta **generator()** pues este método crea y descarta de inmediato un objeto de **FDemo**. En el siguiente paso observará cómo se lleva a cabo esto.

(continúa)

3. Cree la clase **Finalize** que se muestra aquí:

```
class Finalize {
  public static void main(String args[]) {
    int cuenta;

    FDemo ob = new FDemo(0);

    /* Ahora genera gran cantidad de objetos. En
       cierto momento se recolecta la basura.
       Nota: tal vez necesite aumentar la cantidad
       de objetos generados para forzar la
       recolección de basura. */

    for(cuenta=1; cuenta < 100000; cuenta++)
      ob.generator(cuenta);
  }
}
```

Esta clase crea un objeto inicial de **FDemo** llamado **ob**. Luego, usando **ob**, crea 100 000 objetos llamando a **generator( )** sobre **ob**. Esto tiene el efecto neto de crear y descartar 100 000 objetos. En varios puntos a mitad de este proceso, la recolección de basura tiene lugar. Varios factores determinan con precisión la frecuencia y el momento; entre estos factores están la cantidad inicial de memoria libre y el sistema operativo. Sin embargo, en algún punto, usted empezará a ver los mensajes generados por **finalize( )**. Si no ve los mensajes, pruebe a aumentar el número de objetos que se está generando al aumentar la cuenta en el bucle **for**.

4. He aquí el programa **finalize.java** completo:

```
/*
   Proyecto 4-2
   Demuestra el método finalize().
*/

class FDemo {
  int x;

  FDemo(int i) {
    x = i;
  }

  // llamado cuando el objeto se recicla
  protected void finalize() {
    System.out.println("Finaliza " + x);
  }
```

```
    // genera un ob que se destruye de inmediato
    void generator(int i) {
      FDemo o = new FDemo(i);
    }

  }

  class Finalize {
    public static void main(String args[]) {
      int cuenta;

      FDemo ob = new FDemo(0);

      /* Ahora genera gran cantidad de objetos. En
         cierto momento se recolecta la basura.
         Nota: tal vez necesite aumentar la cantidad
         de objetos generados para forzar la
         recolección de basura. */

      for(cuenta=1; cuenta < 100000; cuenta++)
        ob.generator(cuenta);
    }
  }
```

HABILIDAD FUNDAMENTAL 4.12

La palabra clave this

Antes de concluir este módulo, es necesario introducir **this**. Cuando se llama a un método, se pasa automáticamente un argumento implícito que es una referencia al objeto que invoca (es decir, el objeto en el que se llama al método). A esta referencia se le denomina **this**. Para comprender **this**, primero considere un programa que cree una clase llamada **Pot**, la cual calcula el resultado de un número que esté elevado a alguna potencia entera:

```
class Pot {
  double b;
  int e;
  double val;

  Pot(double base, int exp) {
    b = base;
    e = exp;

    val = 1;
    if(exp==0) return;
    for( ; exp>0; exp--) val = val * base;
```

```
  }

  double obtener_pot() {
    return val;
  }
}

class DemoPot {
  public static void main(String args[]) {
    Pot x = new Pot(4.0, 2);
    Pot y = new Pot(2.5, 1);
    Pot z = new Pot(5.7, 0);

    System.out.println(x.b + " elevado a la " + x.e +
                       " potencia es " + x.obtener_pot());
    System.out.println(y.b + " elevado a la " + y.e +
                       " potencia es " + y.obtener_pot());
    System.out.println(z.b + " elevado a la " + z.e +
                       " potencia es " + z.obtener_pot());
  }
}
```

Como ya lo sabe, dentro de un método, es posible acceder directamente a los otros miembros de una clase, sin necesidad de ninguna calificación de objeto o clase; así que, dentro de **obtener_pot()**, la instrucción

```
return val;
```

significa que se devolverá la copia de **val** asociada con el objeto al que invoca. Sin embargo, es posible escribir la misma instrucción de la siguiente manera:

```
return.this.val;
```

En este caso, **this** alude al objeto en el que se llama a **obtener_pot()**. Por consiguiente, **this.val** alude a esa copia del objeto de **val**. Por ejemplo, si se ha invocado a **obtener_pot()** en **x**, entonces **this** en la instrucción anterior hubiera hecho referencia a **x**. Escribir la instrucción sin el uso de **this** constituye en realidad un atajo.

He aquí la clase **Pot** completa, la cual está escrita con la referencia a **this**:

```
class Pot {
  double b;
  int e;
  double val;

  Pot(double base, int exp) {
    this.b = base;
```

```
    this.e = exp;

    this.val = 1;
    if(exp==0) return;
    for( ; exp>0; exp--) this.val = this.val * base;
  }

  double obtener_pot() {
    return this.val;
  }
}
```

En realidad, ningún programador de Java escribiría **Pot** como se acaba de mostrar ya que no se gana nada, así que la forma estándar es más fácil. Sin embargo, **this** tiene algunos usos importantes. Por ejemplo, la sintaxis de Java permite que el nombre de un parámetro o de una variable local sea igual al nombre de una variable de instancia. Cuando esto sucede, el nombre local *oculta* la variable de instancia. Puede obtener acceso a la variable de instancia oculta si hace referencia a ella mediante **this**. Por ejemplo, aunque no se trata de un estilo recomendado, la siguiente es una manera sintácticamente válida de escribir el constructor **Pot()**.

```
Pot(double b, int e) {
  this.b = b;
  this.e = e;
```

Esto se refiere a la variable de instancia **b**, no al parámetro.

```
  val = 1;
  if(e==0) return;
  for( ; e>0; e--) val = val * b;
}
```

En esta versión, los nombres de los parámetros son los mismos que los de las variables de instancia, pero estas últimas están ocultas. Sin embargo, **this** se usa para "descubrir" las variables de instancia.

Comprobación de dominio del módulo 4

1. ¿Cuál es la diferencia entre una clase y un objeto?
2. ¿Cómo se define una clase?
3. ¿De qué tiene cada objeto una copia propia?
4. Empleando dos instrucciones separadas, muestre cómo declarar un objeto llamado **contador** de una clase llamada **MiContador**.

5. Muestre cómo se declara un método llamado **MiMet()** si tiene un tipo de retorno **double** y dos parámetros **int** llamados **a** y **b**.
6. Si un método regresa un valor, ¿cómo debe regresar?
7. ¿Qué nombre tiene un constructor?
8. ¿Qué función tiene **new**?
9. ¿Qué es la recolección de basura y cómo funciona? ¿Qué es **finalize()**?
10. ¿Qué es **this**?
11. ¿Un constructor puede tener uno o más parámetros?
12. Si un método no regresa un valor, ¿cuál debe ser su tipo de regreso?

Módulo 5

Más tipos de datos y operadores

HABILIDADES FUNDAMENTALES

En este módulo se regresa al tema de los tipos de datos y los operadores de Java. En él se analizan las matrices, el tipo **String**, los operadores de bitwise y el operador ternario**?** Asimismo, se estudia el nuevo estilo for-each del bucle **for**. Por cierto, se describen también los argumentos de línea de comandos.

HABILIDAD FUNDAMENTAL 5.1

Matrices

Una matriz es una colección de variables del mismo tipo a las que se hace referencia mediante un nombre común. En Java, las matrices pueden tener una o más dimensiones, si bien la de una dimensión es la más común. Las matrices se usan para diversos propósitos porque ofrecen un medio conveniente de agrupar variables relacionadas. Por ejemplo, tal vez quiera usar una matriz para contener un registro de la temperatura diaria más alta durante un mes, una lista de promedios de precios de acciones o una lista de su colección de libros de programación.

La principal ventaja de una matriz es que organiza los datos de tal manera que pueden manipularse fácilmente. Por ejemplo, si tiene una matriz que contiene los ingresos de un grupo seleccionado de familias, es fácil calcular el ingreso promedio recorriendo la matriz en ciclo. Además, las matrices organizan los datos de tal manera que es posible ordenarlos fácilmente.

Aunque las matrices pueden usarse en Java como en otros lenguajes de programación, éstas se caracterizan por un atributo especial: están implementadas como objetos. Este hecho es una de las razones por las que el análisis de las matrices se pospuso hasta después de que los objetos se presentaron. Al implementar las matrices como objetos, se obtienen varias ventajas importantes. Entre ellas, el hecho de que las matrices que no se utilizan pueden enviarse a la recolección de basura.

Matrices de una dimensión

Una matriz de una dimensión es una lista de variables relacionadas. Estas listas son comunes en programación. Por ejemplo, podría usar una matriz de una dimensión para almacenar los números de cuenta de los usuarios activos en una red. Otra matriz podría usarse para almacenar los promedios actuales de bateo de un equipo de béisbol.

Para declarar una matriz de una dimensión, deberá usar esta forma general:

tipo nombre-matriz[] = new *tipo*[*tamaño*];

En este caso, *tipo* declara el tipo de base de la matriz, la cual determina el tipo de datos de cada elemento contenido en la matriz. El número de elementos contenido en una matriz está determinado por *tamaño*. Debido a que las matrices se implementan como objetos, la creación de una matriz es un proceso de dos pasos. En primer lugar, usted declara una variable de referencia a la matriz. En segundo lugar, asigna memoria a la matriz asignando una referencia a esta memoria en la variable de la matriz. De este modo, las matrices en Java se asignan dinámicamente empleando el operador **new**.

He aquí un ejemplo. El siguiente código crea una matriz **int** de 10 elementos y la vincula con una variable de referencia a matriz llamada **muestra**.

```
int muestra[] = new int[10];
```

Esta declaración funciona como una declaración de objeto. La variable **muestra** contiene una referencia a la memoria asignada por **new**. Esta memoria es suficiente para contener 10 elementos de tipo **int**.

Al igual que como sucede con los objetos, es posible dividir la declaración anterior en dos partes. Por ejemplo:

```
int muestra[];
muestra = new int[10];
```

En este caso, cuando se crea **muestra** por primera vez, es **null** porque no hace referencia a algún objeto físico. Sólo es después de que la segunda instrucción se ejecuta que **muestra** se vincula con una matriz.

Es posible acceder al elemento individual de una matriz mediante el uso de un índice. Un *índice* describe la posición de un elemento dentro de una matriz. En Java, todas las matrices tienen a cero como índice de su primer elemento. Debido a que **muestra** tiene 10 elementos, cuenta con valores de índice de 1 a 9. Para indizar una matriz, especifique el número de elementos que desee entre corchetes. Así, el primer elemento de **muestra** es **muestra[0]** y el último es **muestra[9]**. Por ejemplo, el siguiente programa carga **muestra** con los números del 0 al 9.

```
// Demuestra una matriz de una dimensión.
class MatrizDemo {
  public static void main(String args[]) {
    int muestra[] = new int[10];
    int i;

    for(i = 0; i < 10; i = i+1)
      muestra[i] = i;

    for(i = 0; i < 10; i = i+1)
      System.out.println("Muestra[" + i + "]: " +
                         muestra[i]);
  }
}
```

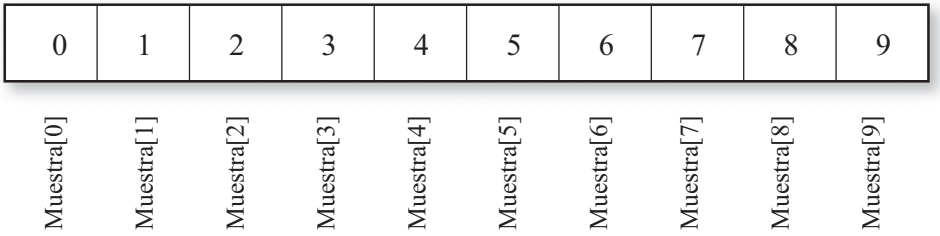
Las matrices están indizadas desde cero.

Ésta es la salida de este programa:

```
Muestra[0]: 0
Muestra[1]: 1
Muestra[2]: 2
Muestra[3]: 3
```

```
Muestra[4]: 4
Muestra[5]: 5
Muestra[6]: 6
Muestra[7]: 7
Muestra[8]: 8
Muestra[9]: 9
```

Conceptualmente, la matriz muestra tiene este aspecto:



Las matrices son comunes en programación porque le permiten tratar fácilmente con números grandes de variables relacionadas. Por ejemplo, el siguiente programa encuentra los valores máximo y mínimo almacenados en la matriz **nums** al recorrer en ciclo la matriz empleando el bucle **for**.

```
// Encuentra los valores mínimo y máximo de una matriz.
class MinMax {
  public static void main(String args[]) {
    int nums[] = new int[10];
    int min, max;

    nums[0] = 99;
    nums[1] = -10;
    nums[2] = 100123;
    nums[3] = 18;
    nums[4] = -978;
    nums[5] = 5623;
    nums[6] = 463;
    nums[7] = -9;
    nums[8] = 287;
    nums[9] = 49;

    min = max = nums[0];
    for(int i=1; i < 10; i++) {
      if(nums[i] < min) min = nums[i];
      if(nums[i] > max) max = nums[i];
    }
    System.out.println("Mínimo y máximo: " + min + " " + max);
  }
}
```

Ésta es la salida del programa:

```
Mínimo y máximo: -978 100123
```

En el programa anterior se asignaron a mano los valores de la matriz **nums** empleando 10 instrucciones de asignación separadas. Aunque esto es perfectamente correcto, hay una manera más fácil de lograrlo. Las matrices pueden inicializarse cuando se crean. A continuación se muestra la forma general para inicializar una matriz de una dimensión:

tipo nombre-matriz[] = { *val1*, *val2*, *val3*,… *valN*];

En este caso, los valores iniciales se especifican de *val1* a *valN*. Están asignados en secuencia, de izquierda a derecha y en orden de índice. Java asigna automáticamente una matriz del largo suficiente para contener los inicializadotes que especifique. No es necesario usar explícitamente el operador **new**. Por ejemplo, he aquí una mejor manera de escribir el programa **MinMax**:

```
// Uso de inicializadores de matriz.
class MinMax2 {
  public static void main(String args[]) {
    int nums[] = { 99, -10, 100123, 18, -978,
                   5623, 463, -9, 287, 49 };
    int min, max;

    min = max = nums[0];
    for(int i=1; i < 10; i++) {
      if(nums[i] < min) min = nums[i];
      if(nums[i] > max) max = nums[i];
    }
    System.out.println("Mínimo y máximo: " + min + " " + max);
  }
}
```

Inicializadores de matriz.

En Java los límites de la matriz se imponen estrictamente. Sobrepasar o quedarse antes del final de una matriz constituye un error en tiempo de ejecución. Si quiere confirmarlo por usted mismo, pruebe el siguiente programa, el cual sobrepasa a propósito una matriz:

```
// Demuestra el sobrepaso de una matriz.
class MatrizErr {
  public static void main(String args[]) {
    int muestra[] = new int[10];
    int i;

    // genera el sobrepaso de la matriz
    for(i = 0; i < 100; i = i+1)
      muestra[i] = i;
  }
}
```

En cuanto **i** llega a 10, una **ArrayIndexOutofBoundsException** se genera y el programa se termina.

Comprobación de avance

1. Se accede a las matrices mediante un _____.
2. ¿Cómo se declara una matriz **char** de 10 elementos?
3. Java no revisa el sobrepaso de una matriz en tiempo de ejecución. ¿Cierto o falso?

Proyecto 5.1 Ordenamiento de una matriz

`Burbuja.java`

Debido a que organiza los datos en una lista lineal indizable, una matriz de una dimensión resulta una estructura de datos ideal para ordenar. En este proyecto aprenderá una manera simple de ordenar una matriz. Como tal vez ya lo sepa, existen varios algoritmos diferentes de orden: los de orden rápido, de agitación y de concha, por nombrar sólo tres. Sin embargo, el más conocido, el más simple y fácil de comprender es el llamado orden de burbuja. Aunque no es muy eficiente (en realidad, su desempeño resulta inaceptable para ordenar matrices grandes), este tipo de algoritmo puede emplearse con efectividad para ordenar matrices pequeñas.

Paso a paso

1. Cree un archivo llamado **Burbuja.java**.
2. El orden de burbuja obtiene su nombre a partir de la manera en la que realiza la operación de ordenamiento: usa la comparación repetida y, si es necesario, el intercambio de elementos adyacentes en la matriz. En este proceso los valores pequeños se mueven hacia un extremo y los grandes hacia el otro. El proceso es conceptualmente similar a las burbujas que encuentran su propio nivel en un tanque de agua. El orden de burbuja opera al llevar a cabo varios pases por la matriz

1. índice
2. char a[ ] = new char[10];
3. Falso. Java no permite que haya sobrepasos de una matriz en tiempo de ejecución.

e intercambiar elementos fuera de lugar cuando es necesario. El número de pases requerido para asegurar que la matriz esté ordenada es igual a uno menos el número de elementos en la matriz.

He aquí el código que forma el eje del orden de burbuja. A la matriz que se está ordenando se le denomina **nums**.

```
// Éste es el orden de burbuja.
for(a=1; a < dimen; a++)
  for(b=dimen-1; b >= a; b--) {
    if(nums[b-1] > nums[b]) { // si está fuera de orden
      // intercambia elementos
      t = nums[b-1];
      nums[b-1] = nums[b];
      nums[b] = t;
    }
  }
```

Observe que el orden depende de dos bucles **for**. El bucle interno revisa los elementos adyacentes en la matriz buscando elementos fuera de orden. Cuando encuentra un par de elementos fuera de orden, los dos elementos se intercambian. Con cada pase, el más pequeño de los elementos restantes se mueve a su ubicación apropiada. El bucle externo hace que el proceso se repita hasta que toda la matriz se haya ordenado.

3. He aquí el programa completo:

```
/*
   Proyecto 5.1
   Demuestra el orden de burbuja.
*/

class Burbuja {
  public static void main(String args[]) {
    int nums[] = { 99, -10, 100123, 18, -978,
                   5623, 463, -9, 287, 49 };
    int a, b, t;
    int dimen;

    dimen = 10; // número de elementos para ordenar

    // despliega la matriz original
    System.out.print("La matriz original es:");
    for(int i=0; i < dimen; i++)
```

(continúa)

```
      System.out.print(« « + nums[i]);
    System.out.println();

    // Éste es el orden de burbuja.
    for(a=1; a < dimen; a++)
      for(b=dimen-1; b >= a; b--) {
        if(nums[b-1] > nums[b]) { // si está fuera de orden
          // intercambia elementos
          t = nums[b-1];
          nums[b-1] = nums[b];
          nums[b] = t;
        }
      }

    // despliega matriz ordenada
    System.out.print("La matriz ordenada es:");
    for(int i=0; i < dimen; i++)
      System.out.print(" " + nums[i]);
    System.out.println();
  }
}
```

La salida del programa es la siguiente:

```
La matriz original es: 99 -10 100123 18 -978 5623 463 -9 287 49
La matriz ordenada es: -978 -10 -9 18 49 99 287 463 5623 100123
```

4. Aunque el orden de burbuja es bueno para matrices pequeñas, no resulta eficiente cuando se usa en matrices grandes. El mejor algoritmo de ordenamiento de propósito general es el de ordenamiento rápido (Quicksort). Sin embargo, éste depende de funciones de Java que no se han explicado aún.

HABILIDAD FUNDAMENTAL 5.2

Matrices de varias dimensiones

Aunque la matriz de una dimensión es la de uso más frecuente en programación, las multidimensionales (matrices de dos o más dimensiones) son también comunes. En Java, una matriz de varias dimensiones es una matriz de matrices.

Matrices de dos dimensiones

La forma más simple de matriz multidimensional es la de dos dimensiones. Se trata, en esencia, de una lista de matrices de una dimensión. Para declarar la matriz entera **tabla**, de dos dimensiones de tamaño 10, 20, tendría que escribir:

```
int tabla[][] = new int[10][20];
```

Preste atención especial a la declaración. A diferencia de otros lenguajes de cómputo, que usan comas para separar las dimensiones de la matriz, Java coloca cada dimensión en su propio conjunto de corchetes. De manera similar, para acceder al punto 3,5 de la matriz **tabla**, tendría que usar: **tabla[3][5]**.

En el siguiente ejemplo, se ha cargado una matriz de dos dimensiones con los números del 1 al 12.

```
// Demuestra una matriz de dos dimensiones.
class DosD {
  public static void main(String args[]) {
    int t, i;
    int tabla[][] = new int[3][4];

    for(t=0; t < 3; ++t) {
      for(i=0; i < 4; ++i) {
        tabla[t][i] = (t*4)+i+1;
        System.out.print(tabla[t][i] + " ");
      }
      System.out.println();
    }
  }
}
```

En este ejemplo, **tabla[0][0]** tendrá el valor 1, **tabla[0][1]** el valor 2, **tabla[0][2]** el valor 3, etc. El valor de **tabla[2][3]** será 12. Conceptualmente, la matriz se parecerá a la mostrada en la figura 5.1.

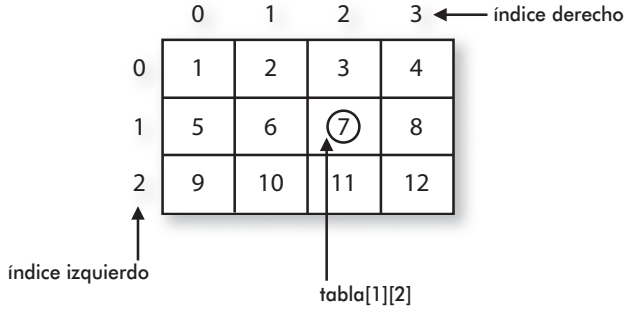


Figura 5.1 Vista conceptual de la matriz de tabla creada por el programa DosD.

HABILIDAD FUNDAMENTAL 5.3

Matrices irregulares

Cuando asigna memoria a una matriz de varias dimensiones, sólo necesita especificar la memoria de la primera dimensión (la del extremo izquierdo). Puede asignar las demás dimensiones por separado. Por ejemplo, el siguiente código asigna memoria a la primera dimensión de **tabla** cuando ésta se declara, y asigna la segunda dimensión manualmente.

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

Si bien asignar individualmente la segunda dimensión de las matrices en esta situación no representa ninguna ventaja, es posible que existan otras ventajas en otras situaciones Por ejemplo, cuando asigna dimensiones por separado, no necesita asignar el mismo número de elementos para cada índice. Debido a que las matrices de varias dimensiones se implementan como matrices de matrices, la longitud de cada matriz está bajo su control. Por ejemplo, suponga que está escribiendo un programa que almacena el número de pasajeros que embarcan en un trasbordador del aeropuerto. Si el trasbordador funciona 10 veces al día entre semana y dos veces al día sábados y domingos, podría usar la matriz **pasajeros** que se muestra en el siguiente programa para almacenar la información. Observe que la longitud de la segunda dimensión para los primeros cinco índices es 10 y la longitud de la segunda dimensión para los últimos dos índices es 2.

```
// Asigna manualmente segundas dimensiones de diferentes tamaños.
class Desigual {
  public static void main(String args[]) {
    int pasajeros[][] = new int[7][];
    pasajeros[0] = new int[10];
    pasajeros[1] = new int[10];
    pasajeros[2] = new int[10];
    pasajeros[3] = new int[10];
    pasajeros[4] = new int[10];
    pasajeros[5] = new int[2];
    pasajeros[6] = new int[2];

    int i, j;

    // fabrica algunos datos falsos
    for(i=0; i < 5; i++)
      for(j=0; j < 10; j++)
        pasajeros[i][j] = i + j + 10;
    for(i=5; i < 7; i++)
      for(j=0; j < 2; j++)
        pasajeros[i][j] = i + j + 10;
```

Aquí las segundas dimensiones son de 10 elementos de largo.

Pero aquí son de 2 elementos de largo.

```
    System.out.println("pasajeros por viaje entre semana:");
    for(i=0; i < 5; i++) {
      for(j=0; j < 10; j++)
        System.out.print(pasajeros[i][j] + " ");
      System.out.println();
    }
    System.out.println();

    System.out.println("pasajeros por viaje el fin de semana:");
    for(i=5; i < 7; i++) {
      for(j=0; j < 2; j++)
        System.out.print(pasajeros[i][j] + " ");
      System.out.println();
    }
  }
}
```

No se recomienda el uso de matrices de varias dimensiones irregulares (o desiguales) para la mayor parte de las aplicaciones porque resulta contrario a lo que la gente esperara encontrar cuando se topa con una matriz de varias dimensiones. Sin embargo, las matrices irregulares pueden usarse de manera efectiva en algunas situaciones. Por ejemplo, si necesita una matriz muy larga de dos dimensiones que apenas esté poblada (es decir, una en la que no se emplearán todos los elementos), una matriz irregular podría ser la solución perfecta.

Matrices de tres o más dimensiones

Java permite matrices de más de dos dimensiones. He aquí la forma general de la declaración de una matriz de varias dimensiones:

tipo nombre[][]...[] = new *tipo*[*tamaño1*][*tamaño2*]...[*tamañoN*];

Por ejemplo, la siguiente declaración crea una matriz entera de tres dimensiones de 4 x10 x 3.

```
int multidim[][][] = new int[4][10][3];
```

Inicialización de matrices de varias dimensiones

Una matriz de varias dimensiones puede inicializarse al incluir la lista de inicializadores de cada dimensión dentro de su propio juego de llaves. Por ejemplo, aquí se muestra la forma general de inicialización de matrices para una matriz de dos dimensiones:

tipo-especificador nombre_matriz[][] = {
  {*val*, *val*, *val*,...,*val*},
  {*val*, *val*, *val*,...,*val*},
.
.
.

```
  {val, val, val,...,val},
};
```

En este caso, *val* indica un valor de inicialización. Cada bloque interno designa una fila. Dentro de cada fila, el primer valor se almacenará en la primera posición de la matriz, el segundo valor en la segunda posición, etc. Observe que las comas separan los bloques inicializadotes y que un punto y coma sigue a la llave de cierre.

Por ejemplo, el siguiente programa inicializa una matriz llamada **cuads** con los números del 1 al 10 y sus cuadrados.

```
// Inicializa una matriz de dos dimensiones.
class Cuadrados {
  public static void main(String args[]) {
    int cuads[][] = {
      { 1, 1 },
      { 2, 4 },
      { 3, 9 },
      { 4, 16 },
      { 5, 25 },
      { 6, 36 },
      { 7, 49 },
      { 8, 64 },
      { 9, 81 },
      { 10, 100 }
    };
    int i, j;

    for(i=0; i < 10; i++) {
      for(j=0; j < 2; j++)
        System.out.print(cuads[i][j] + " ");
      System.out.println();
    }
  }
}
```

Observe como cada fila, tiene su propio grupo de inicializadores.

He aquí la salida del programa:

```
1 1
2 4
3 9
4 16
5 25
6 36
7 49
8 64
9 81
10 100
```

Comprobación de avance

1. En el caso de matrices de varias dimensiones, ¿cómo se especifica cada dimensión?
2. En una matriz de dos dimensiones, la cual es una matriz de matrices, ¿ la longitud de cada matriz puede ser diferente?
3. ¿Cómo se inicializan las matrices de varias dimensiones?

HABILIDAD FUNDAMENTAL 5.4

Sintaxis alterna de declaración de matrices

Es posible emplear una segunda forma para declarar una matriz:

tipo[ ] *nombre-var*

En este caso, los corchetes siguen al especificador de tipo, no al nombre de la variable de la matriz. Por ejemplo, las dos declaraciones siguientes son equivalentes:

```
int contador[] = new int[3];
int[] contador = new int[3];
```

Las dos declaraciones siguientes también son equivalentes:

```
char tabla[][] = new char[3][4];
char[][] tabla = new char[3][4];
```

Esta forma de declaración alterna resulta conveniente cuando se declaran varias matrices al mismo tiempo. Por ejemplo,

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

Esto crea tres variables de matriz de tipo **int**. Es lo mismo que escribir

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

La forma de declaración alterna resulta también útil cuando se especifica una matriz como tipo de retorno de un método. Por ejemplo,

```
int[] unMet() {...
```

Esto declara que **unMet( )** devuelve una matriz de tipo **int**.

1. Cada dimensión se especifica dentro de su propio juego de corchetes.
2. Sí.
3. Las matrices de varias dimensiones se inicializan al poner los inicializadores de cada submatriz dentro de su propio juego de llaves.

HABILIDAD FUNDAMENTAL 5.5

Asignación de referencias a matrices

Como otros objetos, cuando asigna una variable de referencia a matriz a otra matriz, simplemente cambia el objeto al que la variable se refiere y no hace que se lleve a cabo una copia de la matriz, ni que el contenido de una matriz se copie en otra. Por ejemplo, revise el programa:

```
// Asignación de variables de referencia a matriz.
class AsignaRef {
  public static void main(String args[]) {
    int i;

    int nums1[] = new int[10];
    int nums2[] = new int[10];

    for(i=0; i < 10; i++)
      nums1[i] = i;

    for(i=0; i < 10; i++)
      nums2[i] = -i;

    System.out.print("Ésta es nums1: ");
    for(i=0; i < 10; i++)
      System.out.print(nums1[i] + " ");
    System.out.println();

    System.out.print("Ésta es nums2: ");
    for(i=0; i < 10; i++)
      System.out.print(nums2[i] + " ");
    System.out.println();

    nums2 = nums1; // ahora nums2 hace referencia a nums1

    System.out.print("Ésta es nums2 tras asignarse: ");
    for(i=0; i < 10; i++)
      System.out.print(nums2[i] + " ");
    System.out.println();

   // ahora opera en la matriz nums1 a través de nums2
   nums2[3] = 99;

    System.out.print("Ésta es nums1 tras cambiar mediante nums2: ");
```

Asigna una variable de referencia

```
    for(i=0; i < 10; i++)
      System.out.print(nums1[i] + " ");
    System.out.println();
  }
}
```

Ésta es la salida del programa:

```
Ésta es nums1: 0 1 2 3 4 5 6 7 8 9
Ésta es nums2: 0 -1 -2 -3 -4 -5 -6 -7 -8 -9
Ésta es nums2 tras asignarse: 0 1 2 3 4 5 6 7 8 9
Ésta es nums1 tras cambiar mediante nums2: 0 1 2 99 4 5 6 7 8 9
```

Como la salida lo muestra, después de la asignación de **nums1** a **nums2**, ambas variables de referencia a matrices se refieren al mismo objeto.

HABILIDAD FUNDAMENTAL 5.6

Uso del miembro lenght

Debido a que las matrices se implementan como objetos, cada matriz tiene asociada una variable de instancia **lenght** que contiene el número de elementos que la matriz puede contener. He aquí un programa que demuestra esta propiedad:

```
// Uso del miembro de matriz length.
class LengthDemo {
  public static void main(String args[]) {
    int lista[] = new int[10];
    int nums[] = { 1, 2, 3 };
    int tabla[][] = { // una tabla de longitud variable
      {1, 2, 3},
      {4, 5},
      {6, 7, 8, 9}
    };

    System.out.println("La longitud de la lista es " + lista.length);
    System.out.println("La longitud de nums es " + nums.length);
    System.out.println("La longitud de tabla es " + tabla.length);
    System.out.println("La longitud de tabla[0] es " + tabla[0].length);
    System.out.println("La longitud de tabla[1] es " + tabla[1].length);
    System.out.println("La longitud de tabla[2] es " + tabla[2].length);
    System.out.println();
```

```
    // uso de length para inicializar la lista
    for(int i=0; i < lista.length; i++)
      lista[i] = i * i;

    System.out.print("Ésta es la lista: ");
    // ahora usa  length para desplegar la lista
    for(int i=0; i < lista.length; i++)
      System.out.print(lista[i] + " ");
    System.out.println();
  }
}
```

Uso de **lenght** para controlar el bucle **for**.

Este programa despliega la siguiente salida:

```
La longitud de la lista es 10
La longitud de nums es 3
La longitud de tabla es 3
La longitud de tabla[0] es 3
La longitud de tabla[1] es 2
La longitud de tabla[2] es 4

Ésta es la lista: 0 1 4 9 16 25 36 49 64 81
```

Ponga especial atención a la manera en la que se usa **lenght** con la matriz de dos dimensiones **tabla**. Como se explicó, una matriz de dos dimensiones es una matriz de matrices. Por lo tanto, cuando se usa la expresión

```
tabla.lenght
```

obtiene el número de *matrices* almacenadas en **tabla**, que es 3 en este caso. Para obtener la longitud de cualquier matriz individual en **tabla**, debe usar una expresión como la siguiente:

```
tabla(0).lenght
```

que, en este caso, obtiene la longitud a partir de la primera matriz.

Otro elemento que debe observarse en **LenghtDemo** es la manera en la que el bucle **for** empleó **lista.lenght** para regir el número de iteraciones que tiene lugar. Como cada matriz lleva su propia longitud, puede usar esta información en lugar de llevar un registro manual del tamaño de una matriz. Tenga en cuenta que el valor de **lenght** no tiene nada que ver con el número de elementos que estén en uso. Contiene, por otro lado, el número de elementos que la matriz es capaz de contener.

La inclusión del miembro **lenght** simplifica muchos algoritmos al facilitar (y hacer más seguros) ciertos tipos de operaciones con matrices. Por ejemplo, el siguiente programa usa **lenght** para copiar una matriz en otra mientras que evita que una matriz sobrepase los límites y provoque una excepción en tiempo de ejecución.

```
// Uso de la variable length como ayuda para copiar una matriz.
class MCopia {
  public static void main(String args[]) {
    int i;
    int nums1[] = new int[10];
    int nums2[] = new int[10];

    for(i=0; i < nums1.length; i++)
      nums1[i] = i;

    // copia nums1 en nums2
    if(nums2.length >= nums1.length)
      for(i = 0; i < nums2.length; i++)
        nums2[i] = nums1[i];

    for(i=0; i < nums2.length; i++)
      System.out.print(nums2[i] + " ");
  }
}
```

Usa **lenght** para comparar tamaños de matriz.

En este caso, **lenght** ayuda a realizar dos funciones importantes. En primer lugar, se usa para confirmar que la matriz de destino es lo suficientemente grande como para incluir el contenido de la matriz de origen. En segundo lugar, proporciona la condición de terminación del bucle **for** que realiza la copia. Por supuesto, en este ejemplo simple, los tamaños de las matrices se conocen fácilmente, pero este mismo método puede aplicarse a un rango amplio de situaciones más desafiantes.

Comprobación de avance

1. ¿Cómo puede reescribirse lo siguiente?

   ```
   int x[] = new int[10];
   ```

2. Cuando se asigna una referencia a matriz a otra matriz, los elementos de la primera se copian en la segunda. ¿Cierto o falso?

3. En relación con las matrices, ¿qué es **lenght**?

1. int[ ] x = new int[10]
2. Falso. Sólo se cambia la referencia.
3. **lenght** es una variable de instancia que todas las matrices tienen. Contiene el número de elementos que la matriz puede contener.

Proyecto 5.2 Una clase Cola

CDemo.java

Como tal vez ya lo sepa, una estructura de datos es un medio de organizar datos. La estructura más simple de datos es la matriz, que es una lista lineal que soporta el acceso aleatorio a sus elementos. Las matrices suelen usarse como apuntalamiento para estructuras de datos más complejas, como pilas y colas. Una *pila* es una lista en la que se accede a los elementos únicamente en el orden de primero en entrar, último en salir. Una *cola* es una lista en la que se accede a los elementos solamente en el orden de primero en entrar, primero en salir. Por lo tanto, una pila es como una pila de platos en una mesa (el de abajo es el último en usarse). Una cola es como una fila en el banco: el primero en llegar es el primero en ser atendido.

Lo que hace interesantes a las estructuras de datos como pilas y colas es que combinan el almacenamiento de información con los métodos para acceder a esa información. Así pues, las pilas y las colas son *motores de datos* en los que la propia estructura proporciona el almacenamiento y la recuperación, en lugar de que el programa lo realice manualmente. Esta combinación representa, obviamente, una excelente opción para una clase. En este proyecto creará una clase de cola simple.

En general, las colas soportan dos operaciones básicas: colocar y obtener. Cada operación colocar sitúa un nuevo elemento al final de la cola. Cada operación obtener recupera el siguiente elemento del frente de la cola. Las operaciones en una cola son *de consumo*: una vez que un elemento se ha recuperado, ya no se puede recuperar de nuevo. La cola también puede llenarse, si ya no hay espacio para almacenar un elemento más, y puede vaciarse, si se han eliminado todos los elementos.

Un último comentario: hay dos tipos básicos de colas: circular y no circular. Una *cola circular* emplea de nuevo lugares de la matriz cuando se eliminan elementos. Una *cola no circular* no recicla los lugares y con el tiempo se agota. Por razones de espacio, en este ejemplo se crea una cola no circular, pero con un poco de trabajo intelectual y de esfuerzo, puede transformarla fácilmente en una cola circular.

Paso a paso

1. Cree un archivo llamado **CDemo.java**.
2. Aunque hay otras maneras de dar soporte a una cola, el método que usaremos está basado en una matriz, es decir, una matriz proporcionará el almacenamiento para los elementos puestos en la cola. Se accederá a esta matriz mediante dos índices. El índice *colocar* determina el lugar donde se almacenará el siguiente elemento de datos. El índice *obtener* indica el lugar en el que se obtendrá el siguiente elemento de datos. Tenga en cuenta que la operación obtener es de consumo, y que no es posible recuperar el mismo elemento dos veces. Aunque la cola que crearemos almacenará caracteres, la misma lógica puede emplearse para cualquier tipo de objeto. Empiece por crear la clase **Cola** con estas líneas:

```
class Cola {
  char q[]; // esta matriz contiene la cola
  int colocarlug, obtenerlug; // los índices colocar y obtener
```

3. El constructor para la clase **Cola** crea una cola de un tamaño determinado. He aquí el constructor **Cola**:

```
Cola(int dimen) {
  q = new char[dimen+1]; // asigna memoria a la cola
  colocarlug = obtenerlug = 0;
}
```

Observe que, al crearse, la cola es más grande, por uno, que el **tamaño** especificado en dimen. Debido a la manera en la que el algoritmo de cola se implementará, un lugar de la matriz permanecerá sin uso, de modo que la matriz debe crearse en un tamaño más grande, por uno, que el tamaño solicitado para la cola. Los índices colocar y obtener están inicialmente en cero.

4. A continuación se muestra el método **colocar()**, el cual almacena elementos:

```
// coloca un carácter en la cola
void colocar(char ch) {
  if(colocarlug==q.length-1) {
    System.out.println(" -- La cola se ha llenado.");
    return;
  }

  colocarlug++;
  q[colocarlug] = ch;
}
```

El método empieza por revisar la condición de cola llena. Si **colocarlug** es igual al último lugar en la matriz **q**, no habrá más espacio para almacenar elementos. De otra manera **colocarlug** se incrementa y el nuevo elemento se almacena en ese lugar. Por lo tanto, **colocarlug** es siempre el índice del último elemento almacenado.

5. Para recuperar elementos, use el método **obtener()** que se muestra a continuación:

```
// obtiene un carácter de la cola
char obtener() {
  if(obtenerlug == colocarlug) {
    System.out.println(" -- La cola se ha vaciado.");
    return (char) 0;
  }

  obtenerlug++;
  return q[obtenerlug];
}
```

Observe que primero se revisa si la cola está vacía. Si **obtenerlug** y **colocarlug** señalan al mismo elemento, se supone entonces que la cola está vacía. Es por ello que ambos se inicializaron a cero en el constructor **Cola**. A continuación, **obtenerlug** se incrementa y se regresa el siguiente elemento. Por lo tanto, **obtenerlug** indica siempre el lugar del último elemento recuperado.

(continúa)

6. He aquí el programa **CDemo.java** completo:

```
/*
   Proyecto 5-2

   Una clase de cola para caracteres.
*/

class Cola {
  char q[]; // esta matriz contiene la cola
  int colocarlug, obtenerlug; // los índices colocar y obtener

  Cola(int dimen) {
    q = new char[dimen+1]; // asigna memoria a la cola
    colocarlug = obtenerlug = 0;
  }

  // coloca un carácter en la cola
  void colocar(char ch) {
    if(colocarlug==q.length-1) {
      System.out.println(" - La cola se ha llenado.");
      return;
    }

    colocarlug++;
    q[colocarlug] = ch;
  }

  // obtiene un carácter de la cola
  char obtener() {
    if(obtenerlug == colocarlug) {
      System.out.println(" - La cola se ha vaciado.");
      return (char) 0;
    }

    obtenerlug++;
    return q[obtenerlug];
  }
}

// Demuestra la clase Cola.
class CDemo {
  public static void main(String args[]) {
    Cola colaGrande = new Cola(100);
    Cola colaPeque = new Cola(4);
```

```
    char ch;
    int i;

    System.out.println("Uso de colaGrande para almacenar el alfabeto.");
    // coloca algunos números en colaGrande
    for(i=0; i < 26; i++)
      colaGrande.colocar((char) ('A' + i));

    // recupera y despliega elementos de colaGrande
    System.out.print("Contenido de colaGrande: ");
    for(i=0; i < 26; i++) {
      ch = colaGrande.obtener();
      if(ch != (char) 0) System.out.print(ch);
    }

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


    System.out.println("Uso de colaPeque para generar errores.");
    // Ahora, use colaPeque para generar algunos errores
    for(i=0; i < 5; i++) {
      System.out.print("Intento de almacenar " +
                       (char) ('Z' - i));

      colaPeque.colocar((char) ('Z' - i));

      System.out.println();
    }
    System.out.println();

    // más errores en colaPeque
    System.out.print("Contenido de colaPeque: ");
    for(i=0; i < 5; i++) {
      ch = colaPeque.obtener();

      if(ch != (char) 0) System.out.print(ch);
    }
  }
}
```

7. La salida producida por el programa se muestra a continuación:

```
Uso de colaGrande para almacenar el alfabeto.
Contenido de colaGrande: ABCDEFGHIJKLMNOPQRSTUVWXYZ

Uso de colaPeque para generar errores.
```

(continúa)

```
Intento de almacenar Z
Intento de almacenar Y
Intento de almacenar X
Intento de almacenar W
Intento de almacenar V -- La cola se ha llenado.


Contenido de colaPeque: ZYXW - La cola se ha vaciado.
```

8. Por cuenta propia, trate de modificar **Cola** para que almacene otros tipos de objetos. Por ejemplo, haga que almacene variables **int** o **double**.

HABILIDAD FUNDAMENTAL 5.7

El bucle for de estilo for-each

Cuando se trabaja con matrices, es común encontrar situaciones en las que debe examinarse cada elemento de una matriz de principio a fin. Por ejemplo, para calcular la suma de los valores contenidos en una matriz, es necesario examinar cada uno de sus elementos. Lo mismo ocurre cuando se calcula un promedio, se busca un valor, se copia una matriz, etc. Debido a que estas operaciones de "principio a fin" son muy comunes, Java define una segunda forma del bucle **for** que mejora esta operación.

La segunda forma de **for** implementa un bucle de estilo "for-each". Un bucle for-each recorre en ciclo una colección de objetos, como una matriz, en modo estrictamente secuencial, de principio a fin. En años recientes, los bucles for-each han alcanzado popularidad entre los diseñadores de lenguajes de cómputo y entre los programadores. Originalmente, Java no ofrecía un bucle de estilo for-each. Sin embargo, con el lanzamiento de J2SE 5, el bucle **for** se mejoró para proporcionar esta opción. El estilo for-each de **for** también es conocido como *bucle **for** mejorado*. Ambos términos se emplean en este libro.

La forma general del **for** de estilo for-each se muestra a continuación:

for(*tipo var-itr*:*colección*)*bloque-instrucciones*

En este caso, *tipo* especifica el tipo y *var-itr* especifica el nombre de la *variable de iteración* que recibirá los elementos de la colección, de uno en uno, de principio a fin. La colección que se está recorriendo se especifica con *colección*. Son varios los tipos de colecciones que pueden usarse con el **for**, pero el único tipo usado en este libro es la matriz. Con cada iteración del bucle, el siguiente elemento de la colección se recupera y almacena en *var-itr*. El bucle se repite hasta que se han obtenido todos los elementos de la colección. De esta manera, cuando se itera en una matriz de tamaño *N*, el **for** mejorado obtiene los elementos de la matriz en orden de índice, de 0 a *N*-1.

Debido a que la variable de iteración recibe valores de la colección, *tipo* debe ser igual al de los elementos almacenados en la colección, o compatible con ellos. Así, cuando se itera en matrices, *tipo* debe ser compatible con el tipo de base de la matriz.

Pregunte al experto

P: **Aparte de las matrices, ¿qué otros tipos de colecciones puede recorrer el bucle** for **de estilo for-each?**

R: Uno de los usos más importantes del **for** de estilo for-each es recorrer en ciclo el contenido de una colección que esté definido por el Marco Conceptual de Colecciones (Collections Framework). Este último es un conjunto de clases que implementa varias estructuras de datos, como listas, vectores, conjuntos y mapas. Un análisis del Marco Conceptual de Colecciones está más allá del alcance del presente libro, pero puede encontrar mayor información en el libro: *Java: The Complete Reference. J2SE 5 Edition* (McGraw-Hill/Osborne, 2005).

Para comprender la motivación tras un bucle de estilo for-each, considere el tipo de bucle **for** que está diseñado para reemplazar. El siguiente fragmento usa un bucle **for** tradicional para calcular la suma de los valores en una matriz:

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

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

Para calcular la suma, se lee cada elemento de **nums** en orden y de principio a fin; de modo que, toda la matriz se lee en estricto orden secuencial. Esto se logra al indizar manualmente la matriz **nums** por **i**, la variable de control del bucle. Más aún, deben especificarse explícitamente el valor de inicio y fin de la variable de control del bucle, así como su incremento.

El **for** de estilo for-each automatiza el bucle anterior. De manera específica, elimina la necesidad de establecer un contador de bucle, definir un valor de inicio y fin e indizar manualmente la matriz. En cambio, recorre en ciclo y automáticamente toda la matriz, obteniendo un elemento a la vez, en secuencia y de principio a fin. Por ejemplo, a continuación se presenta el fragmento anterior reescrito mediante una versión for-each de **for**:

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

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

Con cada paso por el bucle, se asigna automáticamente a **x** un valor igual al siguiente elemento de **nums**. Por lo tanto, en la primera iteración, **x** contiene 1, en la segunda, **x** contiene 2, etc. No sólo se depura la sintaxis, también se evitan errores de límites.

He aquí un programa completo que demuestra la versión **for**-each del for que se acaba de describir:

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

    // Use for de estilo for-each para desplegar y sumar los valores.
    for(int x : nums) {
      System.out.println("El valor es: " + x);
      suma += x;
    }

    System.out.println("Sumatoria: " + suma);
  }
}
```

Un bucle **for** de estilo for-each.

Ésta es la salida del programa:

```
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
Sumatoria: 55
```

Como esta salida lo muestra, el **for** de estilo for-each recorre en ciclo y automáticamente una matriz de manera secuencial, del índice más bajo al más alto.

Aunque el bucle **for** de estilo for-each itera hasta que se han examinado todos los elementos de una matriz, es posible terminar el bucle antes empleando una instrucción **break**. Por ejemplo, este bucle suma sólo los primeros cinco elementos de **nums**.

```
// Sólo suma los cinco primeros elementos
for(int x : nums) {
  System.out.println("El valor es: " + x);
  suma += x;
  if(x == 5) break;  // detiene el bucle cuando se obtiene 5
}
```

Existe un concepto importante que es necesario comprender acerca del bucle **for** de estilo for-each: su variable de iteración es de "sólo lectura" porque se relaciona con la matriz. Una asignación a la variable de iteración no tiene efecto en la matriz. En otras palabras, no puede cambiar el contenido de la matriz al asignar un nuevo valor a la variable de iteración. Por ejemplo, considere este programa:

```
// El bucle for-each es esencialmente de sólo lectura.
class SinCambio {
  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 cantidades ← Esto no cambia las cantidades.
    }

    System.out.println();

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

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

El primer bucle **for** aumenta el valor de la variable de iteración en un factor de 10. Sin embargo, como lo ilustra el segundo bucle **for**, esta asignación no tiene efecto en la matriz **nums**. La salida, mostrada aquí, lo prueba:

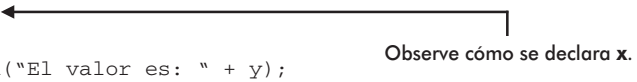
```
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
```

Iteración en matrices de varias dimensiones

El **for** mejorado trabaja sobre matrices de varias dimensiones. Sin embargo, recuerde que en Java las matrices de varias dimensiones están formadas por *matrices de matrices*. (Por ejemplo, una matriz de dos dimensiones es una matriz de matrices de una dimensión.) Esto es importante cuando se itera sobre una matriz de varias dimensiones porque cada iteración obtiene la *siguiente matriz*, no un elemento individual. Más aún, la variable de iteración en el bucle **for** debe ser compatible con el tipo de matriz que se está obteniendo. Por ejemplo, en el caso de una matriz de dos dimensiones, la variable de iteración debe ser una referencia a una matriz de una dimensión. En general, cuando se usa el **for** de estilo for-each para iterar en una matriz de *N* dimensiones, los objetos obtenidos serán matrices de dimensiones *N*-1. Para comprender las implicaciones de esto, considere el siguiente programa, el cual utiliza bucles **for** anidados para obtener los elementos de una matriz de dos dimensiones en orden de fila, de la primera a la última.

Observe cómo se declara **x**.

```
// Uso del estilo for-each para una matriz bidimensional.
class ForEach2 {
  public static void main(String args[]) {
    int suma = 0;
    int nums[][] = new int[3][5];

    // da a nums algunos valores
    for(int i = 0; i < 3; i++)
      for(int j=0; j < 5; j++)
        nums[i][j] = (i+1)*(j+1);

    // Usa for estilo for-each para desplegar y sumar los valores.
    for(int x[] : nums) {
      for(int y : x) {
        System.out.println("El valor es: " + y);
        suma += y;
      }
    }
    System.out.println("Sumatoria: " + suma);
  }
}
```



La salida de este 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: 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
Sumatoria: 90
```

En el programa, preste atención a esta línea:

```
for(int x[] : nums) {
```

Observe cómo se declara **x**. Es una referencia a una matriz de enteros de una dimensión. Esto es necesario porque cada iteración de **for** contiene la siguiente *matriz* en **nums**, empezando con la matriz especificada por **nums[0]**. El bucle **for** interno recorre en ciclo después cada una de estas matrices, desplegando los valores de cada elemento.

Aplicación del for mejorado

Debido a que el **for** de estilo for-each sólo puede recorrer en ciclo una matriz de manera secuencial, de principio a fin, podría pensar que su uso es limitado, pero no es así. Un gran número de algoritmos requieren exactamente este mecanismo. Uno de los más comunes es la búsqueda. Por ejemplo, el siguiente programa usa un bucle **for** para buscar una matriz no ordenada por un valor y se detiene si se encuentra el valor buscado.

```
// Búsqueda de una matriz empleando un for de estilo for-each.
class Buscar {
  public static void main(String args[]) {
    int nums[] = { 6, 8, 3, 7, 5, 6, 1, 4 };
    int val = 5;
    boolean encontrado = false;

    // Use for de estilo for-each para buscar nums por val.
    for(int x : nums) {
      if(x == val) {
        encontrado = true;
        break;
      }
    }

    if(encontrado)
      System.out.println("Valor encontrado");
  }
}
```

El **for** de estilo for-each constituye una elección excelente en esta aplicación porque la búsqueda de una matriz no ordenada incluye el examen de cada elemento en secuencia. (Por supuesto, si la matriz estuviera ordenada, se podría emplear una búsqueda binaria, lo que requeriría un tipo diferente de bucle.) Otros tipos de aplicación que se benefician de los bucles de estilo for-each incluyen el cálculo de un promedio, la búsqueda del mínimo o el máximo de un conjunto, la búsqueda de duplicados, etcétera.

Ahora que se ha introducido el **for** de estilo for-each, éste se utilizará cada vez que resulte apropiado a lo largo del libro.

Comprobación de avance

1. ¿Qué hace un bucle **for** de estilo for-each?
2. Dada una matriz **double** llamada **nums**, muestre un **for** de estilo for-each que la recorra en ciclo.
3. ¿El bucle **for** de estilo for-each puede recorrer en ciclo el contenido de una matriz de varias dimensiones?

HABILIDAD FUNDAMENTAL 5.8

Cadenas

Desde el punto de vista de la programación cotidiana, uno de los tipos de datos más importantes de Java es **String**, el cual define y soporta cadenas de caracteres. En muchos otros lenguajes de programación, una cadena es una matriz de caracteres. No sucede así en Java. En Java, las cadenas son objetos.

En realidad, si bien no lo sabía, ha estado empleando la clase **String** desde el módulo 1. Cuando crea una literal de cadena, en realidad crea un objeto **String**. Por ejemplo, en la instrucción

```
System.out.println("En Java, las cadenas son objetos.");
```

Java convierte la cadena "En Java, las cadenas son objetos." automáticamente en un objeto de **String**. Por lo tanto, el uso de la clase **String** se ha proporcionado "por debajo del agua" en los programas anteriores. En las siguientes secciones aprenderá a manejar esta clase de manera explícita. Sin embargo, tenga cuidado: la clase **String** es muy grande, y aquí sólo rascaremos la superficie. Ésta es una clase que querrá explorar por su cuenta.

Construcción de cadenas

Puede construir una **cadena** de la misma manera que cualquier otro tipo de objeto: empleando **new** y llamando al constructor **String**. Por ejemplo:

```
String cad = new String("Hola");
```

Esto crea un objeto **String** llamado **cad** que contiene la cadena de caracteres "Hola". También puede construir una cadena a partir de otra. Por ejemplo:

```
String cad = new String("Hola");
String cad2 = new String(cad);
```

Después de que se ejecute esta secuencia, **cad2** también contendrá la cadena de caracteres "Hola".

1. Un **for** de estilo for-each recorre en ciclo el contenido de una colección, como una matriz, de principio a fin.
2. for(double d : nums) ...
3. Sí. Sin embargo, cada iteración obtiene la siguiente submatriz.

Aquí se muestra otra manera fácil de crea una **cadena**:

```
String cad = "Las cadenas de Java son importantes.";
```

En este caso, **cad** se inicializa con la secuencia de caracteres "Las cadenas de Java son importantes."

Una vez que ha creado un objeto **String**, puede usarlo en cualquier lugar en el que se permita una cadena entre comillas. Por ejemplo, puede usar un objeto **String** como argumento de **println( )**, como se muestra en este ejemplo:

```
// Introduce String.
class StringDemo {
  public static void main(String args[]) {
    // declara cadenas de varias maneras
    String cad1 = new String("Las cadenas de Java son objetos.");
    String cad2 = "Se construyen de varias maneras.";
    String cad3 = new String(cad2);

    System.out.println(cad1);
    System.out.println(cad2);
    System.out.println(cad3);
  }
}
```

Ésta es la salida del programa:

```
Las cadenas de Java son objetos.
Se construyen de varias maneras.
Se construyen de varias maneras.
```

Operaciones con cadenas

La clase **String** contiene varios métodos que operan sobre cadenas. He aquí unos cuantos:

boolean equals(String *cad*)	Devuelve true si la cadena que invoca contiene la misma secuencia de caracteres que *cad*.
int lenght()	Obtiene la longitud de una cadena
char charAt(int *índice*)	Obtiene el carácter en el índice especificado por *índice*.
int comparteTo(String *cad*)	Regresa menos de cero si la cadena que invoca es menor que *cad*, mayor que cero si la cadena que invoca es mayor que *cad*, y cero si la cadena es igual.
int indexOf(String *cad*)	Busca la subcadena especificada por *cad* en la cadena que invoca. Devuelve el índice de la primera coincidencia o –1 si falla.
int lastIndexOf(String *cad*)	Busca la subcadena especificada por *cad* en la cadena que invoca. Devuelve el índice de la última coincidencia, o –1 si falla.

He aquí un programa que demuestra estos métodos:

```
// Algunas operaciones con String.
class OpsCad {
  public static void main(String args[]) {
    String cad1 =
      "Cuando se programa para Web, Java es la #1.";
    String cad2 = new String(cad1);
    String cad3 = "Las cadenas de Java son importantes.";
    int result, ind;
    char ch;

    System.out.println("La longitud de cad1 es: " +
                       cad1.length());

    // despliega cad1, de carácter en carácter.
    for(int i=0; i < cad1.length(); i++)
      System.out.print(cad1.charAt(i));
    System.out.println();

    if(cad1.equals(cad2))
      System.out.println("cad1 es igual a cad2");
   else
      System.out.println("cad1 no es igual a cad2");

    if(cad1.equals(cad3))
      System.out.println("cad1 es igual a cad3");
    else
      System.out.println("cad1 no es igual a cad3");

    result = cad1.compareTo(cad3);
    if(result == 0)
      System.out.println("cad1 y cad3 son iguales");
    else if(result < 0)
      System.out.println("cad1 es menor que cad3");
    else
      System.out.println("cad1 es mayor que cad3");


    // asigna una nueva cadena a cad2
    cad2 = "Uno Dos Tres Uno";

    ind = cad2.indexOf("Uno");
    System.out.println("Índice de la primera aparición de Uno: " + ind);
    ind = cad2.lastIndexOf("Uno");
    System.out.println("Índice de la última aparición de Uno: " + ind);

  }
}
```

Pregunte al experto

P: **¿Por qué** String **define el método** equals( )**? ¿No podría usar tan sólo ==?**

R: El método **equals( )** compara la secuencia de caracteres de dos objetos **String**. Si se aplica == a las dos referencias a **String**, simplemente se determinará si las dos hacen referencia al mismo objeto.

Este programa genera la siguiente salida:

```
La longitud de cad1 es: 43
Cuando se programa para Web, Java es la #1.
cad1 es igual a cad2
cad1 no es igual a cad3
cad1 es menor que cad3
Índice de la primera aparición de Uno: 0
Índice de la última aparición de Uno: 14
```

Puede *concatenar* (unir) dos cadenas usando el operador +. Por ejemplo, esta instrucción

```
String cad1 = "Uno";
String cad2 = "Dos";
String cad3 = "Tres";
String cad4 = cad1 + cad2 + cad3;
```

inicializa **cad4** con la cadena "Uno Dos Tres".

Matrices de cadenas

Como cualquier otro tipo de datos, las cadenas pueden ensamblarse en matrices. Por ejemplo:

```
// Demuestra matrices de String.
class MatrizCadena {
  public static void main(String args[]) {
    String cads[] = { "Esta", "es", "una", "prueba." };

    System.out.println("Matriz original: ");
    for(String s : cads)
      System.out.print(s + " ");
    System.out.println("\n");
```

Una matriz de cadenas.

```
    // cambia una cadena
    cads[1] = "fue";
    cads[3] = "prueba también!";

    System.out.println("Matriz modificada: ");
    for(String s : cads)
      System.out.print(s + " ");
  }
}
```

He aquí una salida de este programa:

```
Matriz original:
Ésta es una prueba.

Matriz modificada:
Ésta fue una prueba, también!
```

Las cadenas son inmutables

El contenido de un objeto **String** es inmutable. Es decir, una vez creado, la secuencia de caracteres que integra a la cadena no puede modificarse. Esta restricción le permite a Java implementar cadenas de manera más eficiente. Aunque esto probablemente suene como una seria desventaja, no lo es. Cuando necesite una cadena que sea la variación de una que ya exista, simplemente debe crear una nueva cadena que contenga los cambios deseados. Debido a que los objetos **String** se someten automáticamente a la recolección de basura, ni siquiera necesita preocuparse de lo que le sucede a las cadenas descartadas.

Sin embargo, debe quedar claro que las variables de referencia a **String**, por supuesto, cambian el objeto al que hacen referencia. Sólo sucede que el contenido de un objeto **String** específico no puede cambiarse después de creado.

Pregunte al experto

P: **Dice que una vez creados, los objetos** String **son inmutables. Comprendo que, desde un punto de vista práctico, ésta no es una restricción seria, pero, ¿qué pasa si quiero crear una cadena que *sí* pueda cambiarse?**

R: Tiene suerte. Java ofrece una clase llamada **StringBuffer**, la cual crea objetos de cadena que pueden cambiarse. Por ejemplo, además del método **charAt()**, que obtiene el carácter en un lugar específico, **StringBuffer** define **setCharAt()**, que establece un carácter dentro de la cadena. Sin embargo, para casi todo lo que desee utilizará seguramente **String**, no **StringBuffer**.

Para comprender de manera completa por qué las cadenas inmutables no son un impedimento, usaremos otro método de **String**: **substring()**. Este método regresa una nueva cadena que contiene una parte específica de la cadena que invoca. Debido a que hay un nuevo objeto **String** que contiene la subcadena, la cadena original permanece inalterada, mientras que la regla de la inmutabilidad permanece intacta. La forma de **substring()** que usaremos se muestra a continuación:

String substring(int *índiceInicio*, int *índiceFinal*)

Aquí, *índiceInicio* especifica el índice inicial e *índiceFinal* especifica el punto de detención.

Observe el siguiente programa que demuestra **substring()** y el principio de las cadenas inmutables.

```
// Uso de substring().
class SubStr {
  public static void main(String args[]) {
    String cadorig = "Java hace que Web avance.";

    // construye una subcadena
    String subcad = cadorig.substring(5, 18);

    System.out.println("cadorig: " + cadorig);
    System.out.println("subcad: " + subcad);

  }
}
```

Esto crea una nueva cadena que contiene la subcadena deseada.

He aquí la salida del programa:

```
cadorig: Java hace que Web avance.
subcad: hace la Web
```

Como verá, la cadena original **cadorig** permanece sin cambio, y **subcad** contiene la subcadena.

HABILIDAD FUNDAMENTAL

5.9 Uso de argumentos de línea de comandos

Ahora que ya conoce la clase **String**, puede comprender el parámetro **args** de **main()** que ha estado en todos los programas mostrados hasta el momento. Muchos programadores aceptan lo que se ha dado en llamar *argumentos de línea de comandos*. Un argumento de línea de comandos es la información que sigue directamente al nombre del programa en la línea de comandos cuando ésta se ejecuta. Es muy fácil acceder a los argumentos de línea de comandos dentro de un programa de Java: están

almacenados en cadenas en la matriz **String** pasada a **main()**. Por ejemplo, el siguiente programa despliega todos los argumentos de línea de comandos con los que se le llama:

```
// Despliega toda la información de línea de comandos.
class LCDemo {
  public static void main(String args[]) {
    System.out.println("Hay " + args.length +
                       " argumentos de línea de comandos.");

    System.out.println("Son: ");
    for(int i=0; i<args.length; i++)
      System.out.println("arg[" + i + "]: " + args[i]);
  }
}
```

Si se ejecuta **LCDemo** de esta manera,

```
java LCDemo uno dos tres
```

verá la siguiente salida:

```
Hay 3 argumentos de línea de comandos.
Son:
arg[0]: uno
arg[1]: dos
arg[2]: tres
```

Observe que el primer argumento está almacenado en el índice 0, el segundo en el índice 1, etcétera.

Con el fin de que observe la manera en la que pueden usarse los argumentos de la línea de comandos, considere el siguiente programa, el cual toma un argumento de línea de comandos que especifica el nombre de una persona; luego busca ese nombre en una matriz de dos dimensiones de cadena. Si encuentra una coincidencia, despliega el número telefónico de esa persona.

```
// Un directorio telefónico simple automatizado.
class Telef {
  public static void main(String args[]) {
    String tels[][] = {
      { "Juan", "5555-3322" },
      { "Mary", "5555-8976" },
      { "Jaime", "5555-1037" },
      { "Raquel", "5555-1400" }
    };
    int i;
```

```
    if(args.length != 1)
      System.out.println("Uso: java Telef <nombre>");
    else {
      for(i=0; i<tels.length; i++) {
        if(tels[i][0].equals(args[0])) {
          System.out.println(tels[i][0] + ": " +
                             tels[i][1]);
          break;
        }
      }
      if(i == tels.length)
        System.out.println("El nombre no se encuentra.");
    }
  }
}
```

Para usar el programa, un argumento de línea de comandos debe estar presente.

He aquí una ejecución de muestra:

```
C>java Telef Mary
Mary: 5555-8976
```

Comprobación de avance

1. En Java, todas las cadenas son objetos, ¿cierto o falso?
2. ¿Cómo puede obtener la longitud de una cadena?
3. ¿Cuáles son los argumentos de línea de comandos?

HABILIDAD FUNDAMENTAL

5.10 Los operadores de bitwise

En el módulo 2 aprendió acerca de los operadores aritméticos, relacionales y lógicos de Java. Aunque son los de uso más común, Java proporciona operadores adicionales que expanden el conjunto de problemas a los que es posible aplicar Java: los operadores de bitwise. Los operadores de bitwise actúan directamente sobre los bits dentro de los tipos de entero **long**, **int**, **short**, **char** y **byte**. Las

1. Cierto.
2. La longitud de una cadena puede obtenerse si se llama al método **lenght()**.
3. Los argumentos de la línea de comandos se especifican en la línea de comandos cuando un programa se ejecuta y se pasan como cadenas al parámetro **args** de **main()**.

operaciones con bitwise pueden usarse en **boolean**, **float** o **double**. Se les llama operadores de bitwise porque se usan para probar, establecer o desplazar los bits que integran un valor entero. Las operaciones con bitwise son importantes para una amplia variedad de tareas de programación en el nivel del sistema en el que debe pedirse o construirse la información de estado de un dispositivo. En la tabla 5.1 se presenta una lista de los operadores de bitwise.

Los operadores Y, O, XO y NO de bitwise

Los operadores Y, O, XO y NO de bitwise son **&**, |, ^ y ~. Realizan las mismas operaciones que sus equivalentes lógicos booleanos descritos en el módulo 2. La diferencia es que los operadores de bitwise trabajan bit por bit. En la siguiente tabla se muestra el resultado de cada operación empleando 1 y 0.

p	q	p&q	p|q	p^q	~p
0	0	0	0	0	1
1	0	0	1	1	0
0	1	0	1	1	1
1	1	1	1	0	0

En cuanto a su uso más común, puede considerar el Y de bitwise como una manera de inhabilitar los bits. Es decir, cualquier bit que sea 0 en cualquier operando hará que el bit correspondiente en la salida sea 0. Por ejemplo.

```
  1 1 0 1  0 0 1 1
& 1 0 1 0  1 0 1 0
  1 0 0 0  0 0 1 0
```

El siguiente programa demuestra el **&** cuando se convierte cualquier minúscula en mayúscula al cambiar el sexto bit a 0. Tal y como el conjunto de caracteres Unicode/ASCII está definido, las

Operador	Resultado
&	Y de bitwise
|	O de bitwise
^	O excluyente de bitwise
>>	Desplazamiento a la derecha
>>>	Desplazamiento a la derecha sin signo
<<	Desplazamiento a la izquierda
~	Complemento de uno (No unario)

Tabla 5.1 Los operadores de bitwise

minúsculas ocupan el mismo lugar que las mayúsculas, excepto que el valor de las primeras es mayor por 32. Por lo tanto, para transformar una minúscula en mayúscula, sólo debe inhabilitar el sexto bit como se ilustra en el programa.

```
// Letras mayúsculas.
class Mayus {
  public static void main(String args[]) {
    char ch;

    for(int i=0; i < 10; i++) {
      ch = (char) ('a' + i);
      System.out.print(ch);

      // Esta instrucción inhabilita el sexto bit.
      ch = (char) ((int) ch & 65503); // ahora ch es mayúscula

      System.out.print(ch + " ");
    }
  }
}
```

A continuación se muestra la salida de este programa:

```
aA bB cC dD eE fF gG hH iI jJ
```

El valor 65,503 usado en la instrucción Y es la representación decimal de 1111 1111 1101 1111. Por consiguiente, la operación Y deja todos los bits de **ch** sin cambio, excepto por el sexto, que se pone en 0.

El operador Y también resulta útil en el caso de que se quiera determinar si un bit está habilitado o no (si es 1 o 0). Por ejemplo, la siguiente instrucción determina si el bit 4 de **status** está habilitado:

```
if(status & 8) System.out.println("el bit 4 está habilitado");
```

Se usa el número 8 porque se traduce en un valor binario que sólo tiene habilitado el cuarto bit. Por lo tanto, la instrucción **if** sólo puede tener éxito cuando el bit 4 de **status** está también habilitado. Uno de los usos interesantes de este concepto es el de mostrar los bits de un valor **byte** en formato binario.

```
// Despliega los bites dentro de un byte.
class MostrarBits {
  public static void main(String args[]) {
    int t;
    byte val;

    val = 123;
```

```
    for(t=128; t > 0; t = t/2) {
      if((val & t) != 0) System.out.print("1 ");
      else System.out.print("0 ");
    }
  }
}
```

Ésta es la salida:

```
0 1 1 1 1 0 1 1
```

El bucle **for** prueba con éxito cada bit de **val** empleando el Y de bitwise para determinar si está habilitado o no. Si lo está, se despliega el dígito **1**; de lo contrario, se despliega **0**. En el proyecto 5.3 verá cómo este concepto básico puede expandirse para crear una clase que desplegará los bits de cualquier tipo de entero.

El O de bitwise, a diferencia del Y, puede usarse para habilitar un bit. Cualquier bit que esté en 1 en cualquier operando hará que el bit correspondiente en la variable se ponga en 1. Por ejemplo:

```
  1 1 0 1  0 0 1 1
| 1 0 1 0  1 0 1 0
  1 1 1 1  1 0 1 1
```

Podemos usar el O para cambiar el programa de conversión a mayúsculas en uno de conversión a minúsculas como se muestra a continuación:

```
// Letras minúsculas.
class Minus {
  public static void main(String args[]) {
    char ch;

    for(int i=0; i < 10; i++) {
      ch = (char) ('A' + i);
      System.out.print(ch);

      // Esta instrucción habilita el sexto bit.
      ch = (char) ((int) ch | 32); // ahora está en minúsculas

      System.out.print(ch + " ");
    }
  }
}
```

Ésta es la salida de este programa:

```
Aa Bb Cc Dd Ee Ff Gg Hh Ii Jj
```

El programa funciona al someter a O cada carácter con el valor 32, que es 0000 0000 0010 0000 en binario. Por lo tanto, 32 es el valor que produce un valor en binario en el que sólo está establecido el sexto bit. Cuando este valor se somete a O con cualquier otro valor, se produce un resultado en el que el sexto bit se habilita y todos los demás bits permanecen sin cambio. Como se explicó, en el caso de caracteres, esto significa que cada letra mayúscula se transforma en su equivalente en minúsculas.

Un O excluyente, por lo general abreviado XO, habilitará un bit si, y sólo si, los bits comparados son diferentes, como se ilustra a continuación:

```
  0 1 1 1 1 1 1 1
^ 1 0 1 1 1 0 0 1
  ---------------
  1 1 0 0 0 1 1 0
```

El operador de XO tiene una propiedad interesante que simplifica la manera de codificar un mensaje. Cuando algún valor X se somete a XO con otro valor Y, y luego ese resultado se vuelve a someter a XO con Y una vez más, se produce X. Es decir, dada la secuencia

```
R1 = X ^ Y;
R2 = R1 ^ Y;
```

entonces R2 es el mismo valor que X. Así pues, el resultado de una secuencia de dos XO que emplean el mismo valor produce el valor original.

Puede usar este principio para crear un programa simple de cifrado en el que algún entero sea la clave que se utilice para codificar y decodificar un mensaje sometiendo a XO los caracteres en ese mensaje. Para codificar, se aplica la operación XO por primera vez, arrojando el texto cifrado. Para decodificar, se aplica el XO una segunda vez, arrojando el texto simple. He aquí un ejemplo sencillo que utiliza este método para codificar y decodificar un mensaje corto:

```
// Uso de XO para codificar y decodificar un mensaje.
class Codificar {
  public static void main(String args[]) {
    String msj = "Esta es una prueba";
    String codmsj = "";
    String decmsj = "";
    int key = 88;

    System.out.print("Mensaje original: ");
    System.out.println(msj);

    // codifica el mensaje
    for(int i=0; i < msj.length(); i++)
      codmsj = codmsj + (char) (msj.charAt(i) ^ key);

    System.out.print("Mensaje codificado: ");
    System.out.println(codmsj);

    // decodifica el mensaje
```

Esto construye la cadena codificada.

```
    for(int i=0; i < msj.length(); i++)
      decmsj = decmsj + (char) (codmsj.charAt(i) ^ key);
    System.out.print("Mensaje decodificado: ");
    System.out.println(decmsj);
  }
}
```

Esto construye la cadena decodificada.

He aquí la salida:

```
Mensaje original: Ésta es una prueba
Mensaje codificado: 01+x1+x9x, =+,
Mensaje decodificado: Ésta es una prueba
```

Como puede ver, el resultado de los dos XO empleando la misma clave produce el mensaje decodificado.

El operador de complemento (NO) del uno unario opera a la inversa del estado de todos los bits del operando. Por ejemplo, si algún entero llamado A tiene el patrón de bits 1001 0110, entonces ~A produce un resultado con el patrón de bits 0110 1001.

El siguiente programa demuestra el operador NO al desplegar un número y su complemento en binario.

```
// Demuestra el No de bitwise.
class NoDemo {
  public static void main(String args[]) {
    byte b = -34;

    for(int t=128; t > 0; t = t/2) {
      if((b & t) != 0) System.out.print("1 ");
      else System.out.print("0 ");
    }
    System.out.println();

    // revierte todos los bits
    b = (byte) ~b;

    for(int t=128; t > 0; t = t/2) {
      if((b & t) != 0) System.out.print("1 ");
      else System.out.print("0 ");
    }
  }
}
```

He aquí la salida:

```
1 1 0 1 1 1 1 0
0 0 1 0 0 0 0 1
```

Los operadores de desplazamiento

En Java es posible desplazar los bits que integran un valor a la izquierda o la derecha en una cantidad especificada. Java define los tres operadores de desplazamiento de bits que se muestran a continuación:

<<	Desplazamiento a la izquierda
>>	Desplazamiento a la derecha
>>>	Desplazamiento a la derecha sin signo

Éstas son las formas generales de estos operadores:

valor << *num-bits*
valor >> *num-bits*
valor >>> *num-bits*

En este caso, *valor* es el valor que se está desplazando un número de posiciones de bits especificado por *num-bits*.

Cada desplazamiento a la izquierda hace que todos los bits dentro del valor especificado se desplacen una posición a la izquierda y que se lleve un bit 0 a la derecha. Cada desplazamiento a la derecha desplaza una posición todos los bits a la derecha y preserva el bit de signo. Como tal vez ya lo sepa, los números negativos suelen representarse al establecer el bit de orden superior de un valor entero de 1. Por lo tanto, si el valor que se está desplazando es negativo, cada desplazamiento a la derecha trae un 1 a la izquierda. Si el valor es positivo, cada desplazamiento a la derecha trae un 0 a la izquierda.

Además del bit de signo, hay que estar al pendiente de un tema más cuando se desplaza a la derecha. Hoy día, la mayor parte de las computadoras usan el método de *complemento de dos* para valores negativos. En este método, los valores negativos se almacenan al invertir primero los bits en el valor y agregando después 1. De ahí que el valor de bit para –1 en binario sea 1111 1111. El desplazamiento a la derecha de este valor ¡siempre producirá –1!

Si no quiere preservar el bit de signo cuando utilice desplazamiento a la derecha, puede usar un desplazamiento a la derecha sin signo (>>>), el cual siempre produce un 0 a la izquierda. Por ello, a >>> se le denomina el desplazamiento a la derecha de *relleno con ceros*. Usted usará el desplazamiento a la derecha sin signo cuando desplace patrones de bits, como códigos de estado, los cuales no representan enteros.

En todos los desplazamientos, los bits que quedan fuera se pierden. Por lo tanto, un desplazamiento no es una rotación, y no hay manera de recuperar un bit que haya quedado fuera por el desplazamiento.

A continuación se muestra un programa que ilustra gráficamente el efecto de un desplazamiento a la derecha o la izquierda. Aquí, a un entero se le da el valor inicial de 1, que significa que se establece un bit de orden bajo. Luego se realiza una serie de ocho desplazamientos sobre el entero. Después de cada desplazamiento, se muestran los 8 bits inferiores del valor. Luego se repite el valor, excepto que se pone un 1 en la posición del octavo bit y se realizan los desplazamientos a la derecha.

```
// Demuestra los operadores de desplazamiento << y >>.
class DesplDemo {
  public static void main(String args[]) {
```

```
    int val = 1;

    for(int i = 0; i < 8; i++) {
      for(int t=128; t > 0; t = t/2) {
        if((val & t) != 0) System.out.print("1 ");
        else System.out.print("0 ");
      }
      System.out.println();
      val = val << 1; // desplazamiento a la izquierda
    }
    System.out.println();

    val = 128;
    for(int i = 0; i < 8; i++) {
      for(int t=128; t > 0; t = t/2) {
        if((val & t) != 0) System.out.print("1 ");
        else System.out.print("0 ");
      }
      System.out.println();
      val = val >> 1; // desplazamiento a la derecha
    }
  }
}
```

Ésta es la salida del programa:

```
0 0 0 0 0 0 0 1
0 0 0 0 0 0 1 0
0 0 0 0 0 1 0 0
0 0 0 0 1 0 0 0
0 0 0 1 0 0 0 0
0 0 1 0 0 0 0 0
0 1 0 0 0 0 0 0
1 0 0 0 0 0 0 0

1 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0
0 0 1 0 0 0 0 0
0 0 0 1 0 0 0 0
0 0 0 0 1 0 0 0
0 0 0 0 0 1 0 0
0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 1
```

Debe tener cuidado cuando desplace valores **byte** y **short** porque Java promoverá automáticamente estos tipos a **int** cuando evalúe una expresión. Por ejemplo, si desplaza un valor **byte** a la derecha, primero se promoverá a **int** y luego se desplazará. El resultado del desplazamiento será

Pregunte al experto

P: **Como los números binarios están basados en potencias de dos, ¿es posible utilizar operadores de desplazamiento como método abreviado para multiplicar o dividir un entero entre dos?**

R: Sí. Los operadores de desplazamiento de bitwise pueden usarse para realizar con gran rapidez multiplicaciones por dos o divisiones entre dos. Un desplazamiento a la izquierda duplica un valor. Uno a la derecha, lo divide a la mitad. Por supuesto, esto sólo funciona siempre y cuando no esté desplazando bits fuera de un extremo.

de tipo **int**. A menudo esta conversión no implica consecuencias; sin embargo, si desplaza un valor **byte** o **short** negativo, éste se extenderá con el signo cuando se promueva a **int**. Por lo tanto, los bits de orden alto del valor entero resultante se llenarán con unos, lo cual resulta correcto cuando se realiza un desplazamiento normal a la derecha; pero cuando realiza uno de relleno con ceros, se desplazarán 24 unos, antes de que el valor byte empiece a ver ceros.

Asignaciones de método abreviado de bitwise

Todos los operadores binarios de bitwise tienen una forma de método abreviado que combina una asignación con la operación de bitwise. Por ejemplo, las siguientes dos instrucciones asignan a **x** la salida de un XO de **x** con el valor 127.

```
x = x ^ 127;
x ^= 127;
```

Proyecto 5.3 Una clase MostrarBits

`MostrarBitsDemo.java`

Este proyecto crea una clase llamada **MostrarBits** que le permite desplegar en binario el patrón de bits de cualquier valor entero. Esta clase puede ser de gran utilidad en programación. Por ejemplo, si está depurando un código de controlador de dispositivo, entonces la capacidad de monitorear el flujo de datos en binario suele resultar benéfico.

Paso a paso

1. Cree un archivo llamado **MotrarBitsDemo.java.**
2. Inicie la clase **MostrarBits** como se muestra a continuación:

```
class MostrarBits {
  int numbits;
```

(continúa)

```
  MostrarBits(int n) {
    numbits = n;
  }
```

MostrarBits crea objetos que despliegan un número especificado de bits. Por ejemplo, para crear un objeto que despliegue los 8 bits de orden bajo de algún valor, use

```
MostrarBits byteval = new MostrarBits(8)
```

El número de bits que se desplegará está almacenado en **numbits**.

3. Para en realidad desplegar el patrón de bits, **MostrarBits** proporciona el método **mostrar()** que se muestra aquí:

```
void mostrar(long val) {
    long masc = 1;

    // desplaza a la izquierda 1 en la posición apropiada
    masc <<= numbits-1;

    int espaciador = 0;
    for(; masc != 0; masc >>>= 1) {
      if((val & masc) != 0) System.out.print("1");
      else System.out.print("0");
      espaciador++;
      if((espaciador % 8) == 0) {
        System.out.print(" ");
        espaciador = 0;
      }
    }
    System.out.println();
  }
```

Observe que **mostrar()** especifica un parámetro **long**. Sin embargo, esto no significa que un valor **long** siempre tiene que pasar a **mostrar()**. Debido a las promociones automáticas de tipo de Java, cualquier tipo de entero puede pasar a **mostrar()**. El número de bits desplegado está determinado por el valor almacenado en **numbits**. Después de cada grupo de 8 bits, **mostrar()** da salida a un espacio. Esto hace que sea más fácil leer los valores binarios de patrones de bits largos.

4. A continuación se muestra el programa **MostrarBitsDemo**:

```
/*
   Proyecto 5.3

   Una clase que despliega la representación binaria de un valor.
*/

class MostrarBits {
```

```
  int numbits;

  MostrarBits(int n) {
    numbits = n;
  }

  void mostrar(long val) {
    long masc = 1;

    // desplaza a la izquierda 1 en la posición apropiada
    masc <<= numbits-1;

    int espaciador = 0;
    for(; masc != 0; masc >>>= 1) {
      if((val & masc) != 0) System.out.print("1");
      else System.out.print("0");
      espaciador++;
      if((espaciador % 8) == 0) {
        System.out.print(" ");
        espaciador = 0;
      }
    }
    System.out.println();
  }
}

// Demuestra MostrarBits.
class MostrarBitsDemo {
  public static void main(String args[]) {
    MostrarBits b = new MostrarBits(8);
    MostrarBits i = new MostrarBits(32);
    MostrarBits li = new MostrarBits(64);

    System.out.println("123 en binario: ");
    b.mostrar(123);

    System.out.println("\n87987 en binario: ");
    i.mostrar(87987);

    System.out.println("\n237658768 en binario: ");
    li.mostrar(237658768);


    // también puede mostrar bits de orden bajo de cualquier entero
    System.out.println("\n8 bits de orden bajo de 87987 en binario: ");
    b.mostrar(87987);
```

(continúa)

```
    }
}
```

5. Ésta es la salida de **MostrarBitsDemo**:

```
123 en binario:
01111011

87987 en binario:
00000000 00000001 01010111 10110011

237658768 en binario:
00000000 00000000 00000000 00000000 00001110 00101010 01100010 10010000

8 bits de orden bajo de 87987 en binario:
10110011
```

Comprobación de avance

1. ¿A qué tipos pueden aplicarse los operadores de bitwise?

2. ¿Qué es >>>?

HABILIDAD FUNDAMENTAL 5.11

El operador ?

Uno de los operadores más fascinantes es el **?**, que suele usarse para reemplazar instrucciones **if-else** de la siguiente forma general:

if(*condición*)
  var = *expresión1*
else
  var = *expresión2*;

Aquí, el valor asignado a *var* depende de la salida de la condición que controla **if**.

Al **?** se le denomina *operador ternario* porque requiere tres operandos. Adquiere la forma general

Exp1 ? *Exp2* : *Exp3*;

1. **byte**, **short**, **int**, **long** y **char**.
2. >>> realiza un desplazamiento a la derecha sin signo. Esto hace que un cero se desplace en la posición de bit del extremo izquierdo. Se diferencia de >>, el cual preserva el bit de signo.

Donde *Exp1* es una expresión **boolean** y *Exp2* y *Exp3* son expresiones de cualquier tipo diferente de **void**. Sin embargo, los tipos de *Exp2* y *Exp3* deben ser iguales. Observe el uso y la colocación de la coma.

El valor de una expresión **?** se determina evaluando *Exp1*. Si ésta es verdadera, entonces *Exp2* se evalúa y se convierte en el valor de toda la expresión **?**. Si *Exp1* es falsa, entonces *Exp3* se evalúa y su valor se convierte en el valor de toda la expresión. Considere este ejemplo que asigna a **valabs** el valor absoluto de **val**.

```
valabs = val < 0 ? -val : val;  // obtiene el valor absoluto de val
```

Aquí, a **valabs** se le asignará el valor de **val** si **val** es cero o mayor. Si **val** es negativo, entonces a **valabs** se le asignará el negativo de ese valor (el cual arroja un valor positivo). El mismo código escrito con la estructura **if-else** tendría este aspecto:

```
if(val < 0) valabs = -val;
else valabs = val;
```

He aquí otro ejemplo del operador **?**: Este programa divide dos números pero no permite una división entre cero.

```
// Evita una división entre 0 usando ?.
class NoCeroDiv {
  public static void main(String args[]) {
    int result;

    for(int i = -5; i < 6; i++) {
      result = i != 0 ? 100 / i : 0; <-- Esto previene dividir entre cero.
      if(i != 0)
        System.out.println("100 / " + i + " es " + result);
    }
  }
}
```

Ésta es la salida de este programa:

```
100 / -5 es -20
100 / -4 es -25
100 / -3 es -33
100 / -2 es -50
100 / -1 es -100
100 / 1 es 100
100 / 2 es 50
100 / 3 es 33
100 / 4 es 25
100 / 5 es 20
```

Preste especial atención a esta línea del programa:

```
result = i != 0 ? 100 / i : 0;
```

Aquí, a **result** se le ha asignado la salida de la división de 100 entre **i**. Sin embargo, esta división sólo tiene lugar si **i** no es cero. Cuando **i** es cero, se asigna un valor de marcador de posición de cero a **result**.

En realidad no tiene que asignar el valor producido por el **?** a alguna variable. Por ejemplo, podría usar el valor como un argumento en una llamada a un método. O, si todas las expresiones son de tipo **boolean**, el **?** puede usarse como la expresión condicional en un bucle o una instrucción **if**. Por ejemplo, he aquí el programa anterior reescrito de manera un poco distinta. El programa produce la misma salida que antes.

```
// Evita una división entre 0 usando ?
class NoCeroDiv2 {
  public static void main(String args[]) {

    for(int i = -5; i < 6; i++)
      if(i != 0 ? true : false)
        System.out.println("100 / " + i +
                           " is " + 100 / i);
  }
}
```

Observe la instrucción **i**. Si if es cero, entonces la salida de **if** es falsa, se evita la división entre cero y no se despliega un resultado. De otra manera, se realiza la división.

Comprobación de dominio del módulo 5

1. Muestre dos maneras de declarar una matriz de una dimensión de 12 **double**.
2. Muestre cómo inicializar una matriz de una dimensión de enteros a los valores del 1 al 5.
3. Escriba un programa que use una matriz para encontrar el promedio de 10 valores **double**. Use los 10 valores que desee.
4. Cambie el orden del proyecto 5.1 para que ordene una matriz de cadenas. Demuestre que funciona.
5. ¿Cuál es la diferencia entre los métodos de **String indexOf()** y **lastIndexOf()**?
6. Como todas las cadenas son objetos de tipo **String**, muestre cómo puede llamar a los métodos **lenght()** y **charAt()** en esta cadena literal: "Me gusta Java".

7. Al expandir la clase de cifrado **Codificar**, modifíquela para que use una cadena de ocho caracteres como clave.

8. ¿Es posible aplicar los operadores de bitwise al tipo **double**?

9. Muestre cómo puede reescribirse esta secuencia usando el operador **?**

```
if(x < 0) y = 10;
else y = 20;
```

10. En el siguiente fragmento, ¿el operador **&** es lógico o de bitwise? ¿Por qué?

```
boolean a, b;
// ...
if(a & b) ...
```

11. ¿Constituye un error sobrepasar el final de una matriz? ¿Es un error indizar una matriz con un valor negativo?

12. ¿Qué es el operador de desplazamiento a la derecha sin signo?

13. Reescriba la clase **MinMax** que se mostró en este capítulo para que utilice un bucle **for** de estilo for-each.

14. ¿Es posible convertir en bucles de estilo for-each los **for** que realizan el ordenamiento en la clase **Burbuja** que se mostró en el proyecto 5.1? Si no es así, ¿por qué?

Módulo 6

Un análisis detallado de métodos y clases

HABILIDADES FUNDAMENTALES

En este módulo se presenta un resumen de nuestro examen de las clases y los métodos. Se inicia con una explicación sobre cómo controlar el acceso a los miembros de una clase. Luego se analiza el paso y el regreso de objetos, la sobrecarga de métodos, la recursión y el uso de la palabra clave **static**. Asimismo, se describen las clases anidadas y los argumentos de longitud variable, una de las más recientes características de Java.

HABILIDAD FUNDAMENTAL 6.1

Control de acceso a miembros de clases

En su soporte de encapsulamiento, la clase proporciona dos beneficios importantes. En primer lugar, vincula los datos con el código que los manipula (ya ha aprovechado este aspecto de la clase desde el módulo 4). En segundo lugar, proporciona los medios mediante los cuales se puede controlar el acceso a los miembros (esta función es la que se examinará aquí).

Aunque el método de Java es un poco más complejo, en esencia hay dos tipos básicos de miembros de clase: públicos y privados. Se puede acceder libremente a un miembro público, el cual hemos utilizado hasta el momento mediante un código definido fuera de su clase. Se puede acceder a un miembro privado sólo mediante otros métodos definidos por su clase. El acceso se controla mediante el uso de miembros privados.

La restricción al acceso de los miembros de una clase constituye una parte fundamental de la programación orientada a objetos porque contribuye a evitar el mal uso de un objeto. Al permitir el acceso a los datos privados sólo mediante un conjunto de métodos bien definidos, se evita que se asignen valores inapropiados a esos datos (por ejemplo, al comprobar el rango). No es posible que el código fuera de la clase establezca directamente el valor de un miembro privado. También puede controlar con precisión la manera y el momento en que se usan los datos dentro de un objeto. Así que, cuando se implementa correctamente, una clase crea una "caja negra" que puede utilizarse, pero de la cual no se conoce el funcionamiento interno.

Hasta este momento, no ha tenido que preocuparse por el control de acceso pues Java proporciona un parámetro de acceso predeterminado en el que los miembros de una clase están libremente disponibles para el resto del código de su programa. Por consiguiente, el valor de acceso predeterminado es esencialmente público. Aunque resulta conveniente para clases simples (y programas de ejemplo en libros como éste), dicho parámetro predeterminado resulta inadecuado para muchas situaciones reales. Aquí verá cómo usar otras funciones de control de acceso de Java.

Especificadores de acceso de Java

El control de acceso a miembros se logra mediante el uso de tres *especificadores*: **public**, **private** y **protected**. Como ya se explicó, si no se emplea un especificador de acceso, se supone entonces que es necesario utilizar el parámetro de acceso predeterminado. En este módulo nos encargaremos de **public** y **private**. El especificador **protected** sólo se aplica cuando la herencia interviene; además, se describirá en el módulo 8.

Cuando un miembro de una clase se modifica con el especificador **public**, el resto del código del programa puede acceder a ese miembro. Esto incluye métodos definidos dentro de otras clases.

Cuando un miembro de una clase como **private** se especifica, ese miembro sólo puede ser accedido por otros miembros de su clase. Por lo tanto, los métodos de otras clases no pueden acceder al miembro **private** de otra clase.

El parámetro de acceso predeterminado (en el que no se usa un especificador de acceso) es el mismo que **public**, a menos que su programa esté dividido en paquetes. Un *paquete* es, en esencia, un agrupamiento de clases. Los paquetes constituyen una función de organización y de control de acceso, pero no es sino hasta el módulo 8 que se estudiará el análisis de los paquetes. Para los tipos de programa que se muestran en éste y en los módulos anteriores, el acceso **public** es el mismo que el acceso predeterminado.

Un especificador de acceso precede al resto de la especificación de tipo de un miembro. Es decir, debe iniciar una instrucción de declaración de un miembro. He aquí algunos ejemplos:

```
public String mjsErr;
private saldoCuenta sal;

private boolean esError(estado de byte) { // ...
```

Para comprender los efectos de **public** o **private**, considere el siguiente programa:

```
// Acceso public vs private.
class MiClase {
  private int alfa; // acceso private
  public int beta; // acceso public
  int gamma; // acceso predeterminado (en esencia public)

  /* Métodos para acceder a alfa. Está bien que un
     miembro de una clase acceda a un miembro private
     de la misma clase.
  */
  void estableceralfa(int a) {
    alfa = a;
  }

  int obteneralfa() {
    return alfa;
  }
}

class AccesoDemo {
  public static void main(String args[]) {
    MiClase ob = new MiClase();
```

```
    /* El acceso a alfa sólo se permite mediante
       sus métodos de acceso. */
    ob.estableceralfa(-99);
    System.out.println("ob.alfa es " + ob.obteneralfa());

    // No puede acceder a alfa así:
//  ob.alfa = 10; // ¡Incorrecto! ¡alfa es private!  <-- Incorrecto, alfa es private.

    // Éstas están bien porque beta y gamma son public.
    ob.beta = 88;   <-- OK porque son public.
    ob.gamma = 99;
   }
}
```

Como puede ver, dentro de la clase **MiClase**, **alfa** está especificado como **private**, **beta** está explícitamente especificado como **public** y **gamma** usa el acceso predeterminado que, para este ejemplo, es el mismo que si se especificara **public**. Debido a que **alfa** es private, el código que está fuera de la clase no puede accederlo directamente. Debe accederse mediante sus métodos de accesor público: **estableceralfa()** y **obteneralfa()**. Si eliminara el símbolo de comentario al principio de la siguiente línea:

```
//  ob.alfa = 10; // ¡Incorrecto! ¡alfa es private!
```

no podría compilar este programa debido a una violación de acceso. Aunque el acceso a **alfa** por parte del código externo a **MiClase** no está permitido, es posible acceder libremente a los métodos definidos dentro de **MiClase**, como lo muestran los métodos **estableceralfa()** y **obteneralfa()**.

La clave aquí es: otros miembros de su clase pueden usar un miembro private, pero el código que está fuera de su clase no lo puede usar.

Para ver cómo el control de acceso puede aplicarse a un ejemplo más práctico, considere el siguiente programa que implementa una matriz **int** de "falla suave", en el que se evitan los errores de límite y con lo que se evita también una excepción en tiempo de ejecución. Esto se logra al encapsular la matriz como un miembro private de una clase, permitiendo sólo el acceso a la matriz mediante los métodos del miembro. Con este método, se evita cualquier intento de acceder a la matriz más allá de sus límites. Este intento fallará de manera decorosa (lo que dará como resultado un aterrizaje "suave" en lugar de un "choque"). La matriz de falla suave se implementa con la clase **MatrizFallaSuave** que se muestra aquí:

```
/* Esta clase implementa una matriz de "falla suave"
   que evita errores en tiempo de ejecución.
 */
class MatrizFallaSuave {
  private int a[]; // referencia a matriz
  private int valerr; // valor que se regresa si obtener() falla
```

```
  public int length; // length es public

  /* Construye la matriz proporcionando su tamaño y el valor que se regresa
si obtener() falla. */
  public MatrizFallaSuave(int dimen, int errv) {
    a = new int[dimen];
    valerr = errv;
    length = dimen;
  }

  // Regresa el valor del índice dado.
  public int obtener(int indice) {
    if(ok(índice)) return a[índice];
    return valerr;
  }

  // Pone un valor en un índice. Regresa false si falla.
  public boolean colocar(int índice, int val) {
    if(ok(índice)) {
      a[índice] = val;
      return true;
    }
    return false;
  }

  // Regresa true si índice está dentro de los límites.
  private boolean ok(int índice) {
   if(índice >= 0 & índice < length) return true;
   return false;
  }
}

// Demuestra la matriz de falla suave.
class FSDemo {
  public static void main(String args[]) {
    MatrizFallaSuave fs = new MatrizFallaSuave(5, -1);
    int x;

    // Muestra fallas silenciosas
    System.out.println("Falla tranquilamente.");
    for(int i=0; i < (fs.length * 2); i++)
      fs.colocar(i, i*10);

    for(int i=0; i < (fs.length * 2); i++) {
      x = fs.obtener(i);
```

Atrapa un índice fuera de límites.

El acceso a la matriz debe llevarse a cabo con los métodos de acceso.

```
      if(x != -1) System.out.print(x + " ");
    }
    System.out.println("");

    // ahora, maneja las fallas
    System.out.println("\nFalla con reportes de error.");
    for(int i=0; i < (fs.length * 2); i++)
      if(!fs.colocar(i, i*10))
        System.out.println("índice " + i + " fuera de límites");

    for(int i=0; i < (fs.length * 2); i++) {
      x = fs.obtener(i);
      if(x != -1) System.out.print(x + " ");
      else
        System.out.println("índice " + i + " fuera de límites");
    }
  }
}
```

A continuación se muestra la salida del programa:

```
Falla tranquilamente.
0 10 20 30 40

Falla con reportes de error.
Índice 5 fuera de límites
Índice 6 fuera de límites
Índice 7 fuera de límites
Índice 8 fuera de límites
Índice 9 fuera de límites
0 10 20 30 40 Índice 5 fuera de límites
Índice 6 fuera de límites
Índice 7 fuera de límites
Índice 8 fuera de límites
```

Revisemos cuidadosamente este ejemplo. Dentro de **MatrizFallaSuave** están definidos tres miembros **private**. El primero es **a**, que almacena una referencia a la matriz que contiene realmente la información. La segunda es **valerr**, que es el valor que se devolverá cuando una llamada a **obtener()** falla; la tercera es el método **private ok()**, que determina si un índice se encuentra dentro de los límites. Los otros miembros sólo pueden usar estos tres miembros de la clase **MatrizFallaSuave**. De manera específica, **a** y **valerr** sólo pueden ser utilizadas por otros métodos de la clase, mientras que

ok() sólo puede ser llamada por otros miembros de **MatrizFallaSuave**. El resto de los miembros de clase son **public** y pueden ser llamados desde cualquier otro código de un programa que use **MatrizFallaSuave**.

Cuando se construye un objeto **MatrizFallaSuave**, se debe especificar el tamaño de la matriz y el valor que quiere que se regrese si una llamada a **obtener()** falla. El valor de error debe ser un valor que de otra manera no se almacenaría en la matriz. Una vez construidos, tanto la matriz actual a la que hace referencia **a** como el valor de error almacenado en **valerr** no pueden ser accedidos por los usuarios del objeto **MatrizFallaSuave**. De esta manera no se encuentran expuestos a un mal uso. Por ejemplo, el usuario no puede tratar de indizar **a** directamente, con lo que tal vez se excederían sus límites. El acceso sólo está disponible a través de los métodos **obtener()** y **colocar()**.

El método **ok()** es **private** principalmente para cumplir con el objetivo del ejemplo. Hacerlo **public** resultaría inofensivo porque no modifica al objeto. Sin embargo, como la clase **MatrizFallaSuave** utiliza el método internamente, éste puede ser **private**.

Observe que la variable de instancia **lenght** es **public**, lo cual tiene la intención de ser consistente con la manera en que Java implementa las matrices. Para obtener la longitud de una matriz **MatrizFallaSuave**, simplemente utilice su miembro **lenght**.

Para usar una matriz **MatrizFallaSuave**, llame a **colocar()** con el fin de almacenar un valor en el índice especificado. Llame a **obtener()** para recuperar un valor del índice especificado. Si el índice está fuera de los límites, **colocar()** devuelve **false** y **obtener()** devuelve **valerr**.

Por conveniencia, la mayor parte de los ejemplos presentados en este libro seguirán usando el acceso predeterminado para la mayor parte de los miembros. Sin embargo, recuerde que, en la realidad, la restricción de acceso a miembros (especialmente variables de instancia) constituye una parte importante de la programación orientada a objetos que se aplica con éxito. Como verá en el módulo 7, el control de acceso resulta aún más vital cuando se relaciona con la herencia.

Comprobación de avance

1. Nombre los especificadores de acceso de Java.
2. Explique la función de **private**.

1. **private, public** y **protected**. Un acceso predeterminado está también disponible.
2. Cuando un miembro se especifica como **private**, sólo otros miembros de su clase pueden tener acceso a él.

Proyecto 6.1 Mejoramiento de la clase Cola

Cola.java

Puede usar el especificador **private** para realizar una mejora significativamente importante a la clase **Cola** que se desarrolló en el proyecto 5.2 del módulo 5. En esa versión, todos los miembros de la clase **Cola** utilizan el acceso predeterminado que, en esencia, es público. Esto significa que es posible que un programa use **Cola** para acceder de manera directa a la matriz, con lo que tal vez accedería a sus elementos fuera de turno. Debido a que lo importante de una cola es proporcionar una lista del tipo primero en entrar, primero en salir, no se recomienda permitir el acceso fuera de orden. También es posible que un programador avieso modifique los valores almacenados en los índices **colocarlug** y **obtenerlug**, lo que corrompería la cola. Por fortuna, es fácil evitar este tipo de problemas si se aplica el especificador **private**.

Paso a paso

1. Copie la clase **Cola** original del proyecto 5.2 en un nuevo archivo llamado **Cola.java**.
2. En la clase **Cola**, agregue el especificador **private** a la matriz **q** y a los índices **colocarlug** y **obtenerlug** como se muestra a continuación:

```
/*
   Proyecto 6.1
   Una clase mejorada de cola para caracteres.
class Cola {
  // Ahora estos miembros son privados
  private char q[]; // esta matriz contiene la cola
  private int colocarlug, obtenerlug; // los índices colocar y obtener

  Cola(int dimen) {
    q = new char[dimen+1]; // asigna memoria a la cola
    colocarlug = obtenerlug = 0;
  }

  // coloca un carácter en la cola
  void colocar(char ch) {
    if(colocarlug==q.length-1) {
      System.out.println(" -- La cola se ha llenado.");
      return;
    }

    colocarlug++;
    q[colocarlug] = ch;
  }

  // obtiene un carácter de la cola
```

```
  char obtener() {
    if(obtenerlug == colocarlug) {
      System.out.println(" - La cola se ha vaciado.");
      return (char) 0;
    }

    obtenerlug++;
    return q[obtenerlug];
  }
}
```

3. El cambio de **q**, **colocarlug** y **obtenerlug**, de acceso predeterminado a private, no tiene efecto en un programa que utilice apropiadamente **Cola**. Por ejemplo, éste funciona aún con la clase **CDemo** del proyecto 5.2; sin embargo, evita el uso inapropiado de **Cola**. Los siguientes tipos de instrucciones, por ejemplo, son ilegales:

```
Cola prueba = new Cola(10);

prueba.q(0) = 99;  // ¡Incorrecto!
prueba.colocarlug = -100;  // ¡No funciona!
```

4. Ahora que **q**, **colocarlug** y **obtenerlug** son private, la clase **Cola** impone estrictamente el atributo primero en entrar, primero en salir, de una cola.

HABILIDAD FUNDAMENTAL 6.2

Paso de objetos a métodos

Hasta el momento, los ejemplos de este libro han empleado tipos simples como parámetros de métodos. Sin embargo, resulta correcto y común pasar objetos a métodos. Por ejemplo, el siguiente programa define una clase llamada **Bloque** que almacena las dimensiones de un bloque de tres dimensiones:

```
// Pueden pasarse objetos a métodos.
class Bloque {
  int a, b, c;
  int volumen;

  Bloque(int i, int j, int k) {
    a = i;
    b = j;
    c = k;
    volumen = a * b * c;
  }
```

```
  // Devuelve true si ob define el mismo bloque.
  boolean mismoBloque(Bloque ob) {
    if((ob.a == a) & (ob.b == b) & (ob.c == c)) return true;
    else return false;
  }

  // Regresa true si ob tiene el mismo volumen.
  boolean mismoVolumen(Bloque ob) {
    if(ob.volumen == volumen) return true;
    else return false;
  }
}

class PasaOb {
  public static void main(String args[]) {
    Bloque ob1 = new Bloque(10, 2, 5);
    Bloque ob2 = new Bloque(10, 2, 5);
    Bloque ob3 = new Bloque(4, 5, 5);


    System.out.println("ob1 mismas dimensiones que ob2: " +
                       ob1.mismoBloque(ob2));
    System.out.println("ob1 mismas dimensiones que ob3: " +
                        ob1.mismoBloque(ob3));
    System.out.println("ob1 mismo volumen que ob3: " +
                       ob1.mismoVolumen(ob3));
  }
}
```

Usa tipo de objeto para parámetro.

Pasa un objeto.

Este programa genera la siguiente salida:

```
ob1 mismas dimensiones que ob2: true
ob1 mismas dimensiones que ob3: false
ob1 mismo volumen que ob3: true
```

Los métodos **mismoBloque( )** y **mismoVolumen( )** comparan el objeto **Bloque** pasado como parámetro al objeto que invoca. Para **mismoBloque( )** las dimensiones de los objetos se comparan y **true** se regresa si los dos bloques son iguales. En el caso de **mismoVolumen( )**, los dos bloques se comparan sólo para determinar si tienen el mismo volumen. En ambos casos, observe que el parámetro **ob** especifica **Bloque** como su tipo. Aunque **Bloque** es un tipo de clase creado por el programa, se usa de la misma manera que los tipos integrados de Java.

Cómo se pasan los argumentos

Como se demostró en el ejemplo anterior, el paso de un objeto a un método constituye una tarea sencilla. Sin embargo, existen ciertos inconvenientes en pasar un objeto, mismos que no se muestran en el ejemplo. En ciertos casos, los efectos de pasar un objeto serán diferentes a los experimentados cuando se pasan argumentos que no son objetos. Para entender la razón de ello, necesita comprender las dos maneras en las que un argumento puede pasarse a una subrutina.

La primera manera es mediante una *llamada por valor*. Este método copia el *valor* de un argumento en el parámetro formal de la subrutina. Por lo tanto, los cambios realizados al parámetro de la subrutina no tienen efecto en el argumento de la llamada. La segunda manera en la que puede pasarse un argumento es mediante una *llamada por referencia*. En este método, se pasa una referencia a un argumento (no el valor del argumento) al parámetro. Dentro de la subrutina, esta referencia se emplea para acceder al argumento actual especificado en la llamada. Esto significa que los cambios hechos al parámetro *afectarán* al argumento empleado para llamar a la subrutina. Como verá, Java usa ambos métodos, dependiendo de lo que se pase.

En Java, cuando un tipo primitivo, como **int** o **double**, se pasa a un método, se pasa por valor. Por consiguiente, lo que le ocurre al parámetro que recibe el argumento no tiene efecto fuera del método. Considere, por ejemplo, el siguiente programa:

```
// Los tipos simples se pasan por valor.
class Prueba {
  /* Este método no le genera cambios al argumento
     usado en la llamada. */
  void NoCambio(int i, int j) {
    i = i + j;
    j = -j;
  }
}

class LlamadaPorValor {
  public static void main(String args[]) {
    Prueba ob = new Prueba();

    int a = 15, b = 20;

    System.out.println("a y b antes de la llamada: " +
                       a + " " + b);

    ob.NoCambio(a, b);
```

```
    System.out.println("a y b tras la llamada: " +
                       a + " " + b);
  }
}
```

Ésta es la salida de este programa:

```
a y b antes de la llamada: 15 20
a y b tras la llamada: 15 20
```

Como puede ver, la operación que ocurren dentro de **NoCambio()** no tiene efecto en los valores de **a** y **b** usados en la llamada.

Cuando pasa un objeto a un método, la situación cambia dramáticamente, porque los objetos se pasan implícitamente como referencia. Tenga en cuenta que cuando crea una variable de un tipo de clase, sólo está creando una referencia a un objeto. Por tanto, cuando pasa esta referencia a un método, el parámetro que lo recibe hará referencia al mismo objeto que hizo referencia el argumento. Esto significa, efectivamente, que los objetos son pasados a los métodos mediante el uso de la llamada por referencia. Los cambios al objeto dentro del método *sí* afectan al objeto usado como argumento. Por ejemplo, considere el siguiente programa:

```
// Los objetos se pasan por referencia.
class Prueba {
  int a, b;

  Prueba(int i, int j) {
    a = i;
    b = j;
  }
  /* Pasa un objeto. Ahora, serán cambiados en la llamada·n ob.a y ob.b
     usados. */
  void cambio(Prueba ob) {
    ob.a = ob.a + ob.b;
    ob.b = -ob.b;
  }
}

class LlamadaPorRef {
  public static void main(String args[]) {
    Prueba ob = new Prueba(15, 20);

    System.out.println("ob.a y ob.b antes de la llamada: " +
                       ob.a + " " + ob.b);

    ob.cambio(ob);
```

```
    System.out.println("ob.a y ob.b tras 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 tras la llamada: 35 -20
```

Como puede ver, en este caso las acciones dentro de **cambio()** han afectado al objeto utilizado como argumento.

Como punto de interés, cuando se pasa a un método una referencia a objeto, la propia referencia se pasa mediante el uso de una llamada por valor. Sin embargo, como el valor que se está pasando hace referencia a un objeto, la copia de ese valor hará referencia aún al mismo objeto al que el argumento correspondiente hace referencia.

Pregunte al experto

P: **¿Existe alguna forma en la que pueda pasar un tipo primitivo por referencia?**

R: No directamente. Sin embargo, Java define un conjunto de clases que envuelven los tipos primitivos en objetos: **Double, Float, Byte, Short, Integer, Long** y **Character**. Además de permitir que un tipo primitivo se pase por referencia, estas clases de envoltura definen varios métodos que le permiten manipular sus valores. Por ejemplo, las envolturas numéricas de tipo incluyen métodos que hacen que un valor numérico cambie de su forma binaria a su forma String legible para una persona, y viceversa.

Comprobación de avance

1. ¿Cuál es la diferencia entre llamada por valor y llamada por referencia?
2. ¿Cómo pasa Java los tipos primitivos? ¿Cómo pasa los objetos?

1. En una llamada por valor, una copia del argumento pasa a una subrutina. En una llamada por referencia, se pasa una referencia al argumento.
2. Java pasa tipos primitivos por valor y tipos de objetos por referencia.

HABILIDAD FUNDAMENTAL 6.3

Regreso de objetos

Un método puede regresar cualquier tipo de datos, incluyendo tipos de clases. Por ejemplo, la clase **MsjErr** mostrada aquí podría usarse para reportar errores. Su método **obtenerMsjErr( )** devuelve un objeto **String** que contiene una descripción de un error basado en el código de error pasado.

```
// Regresa un objeto String.
class MsjError {
  String msjs[] = {
    "Error de salida",
    "Error de entrada",
    "Disco lleno",
    "Índice fuera de límites"
  };

  // Regresa el mensaje de error.
  String obtenerMsjError(int i) {
    if(i >=0 & i < msjs.length)
      return msjs[i];
    else
      return "El código de error no existe";
  }
}

class MsjErr {
  public static void main(String args[]) {
    MsjError err = new MsjError();

    System.out.println(err.obtenerMsjError(2));
    System.out.println(err.obtenerMsjError(19));
  }
}
```

Devuelve un objeto de tipo **String**.

Aquí se muestra su salida:

```
Disco lleno
El código de error no existe
```

Por supuesto, también puede regresar objetos creados por usted. Por ejemplo, he aquí una versión retrabajada del programa anterior que crea dos clases de error. Uno llamado **Err**, que encapsula un mensaje de error junto con un código de severidad, y el segundo, **InfoError**, el cual define un método llamado **obtenerInfoError( )** que devuelve un objeto **Err**.

```
// Regresa un objeto definido por el programador.
class Err {
  String msj; // mensaje de error
  int severidad; // código que indica severidad del error
   Err(String m, int s) {
    msj = m;
    severidad  = s;
  }
}

class InfoError {
  String msjs[] = {
    "Error de salida",
    "Error de entrada",
    "Disco lleno",
    "Índice fuera de límites"
  };
  int malo[] = { 3, 3, 2, 4 };

  Err obtenerInfoError(int i) {
    if(i >=0 & i < msjs.length)
      return new Err(msjs[i], malo[i]);
    else
      return new Err("El código de error no existe", 0);
  }
}

class InfoErr {
  public static void main(String args[]) {
    InfoError err = new InfoError();
    Err e;

    e = err.obtenerInfoError(2);
    System.out.println(e.msj + " severidad: " + e.severidad);

    e = err.obtenerInfoError(19);
    System.out.println(e.msj + " severidad: " + e.severidad);
  }
}
```

Regresa un objeto de tipo **Err**.

He aquí la salida:

```
Disco lleno severidad: 2
El código de error no existe severidad: 0
```

Cada vez que se invoca **obtenerInfoError()**, se crea un nuevo objeto **Err** y se regresa una referencia a él en la rutina de llamada. Luego se utiliza este objeto dentro de **main()** para desplegar el mensaje de error y el código de severidad.

Cuando un método regresa un objeto, sigue existiendo hasta que ya no hay más referencias a él. En este punto es cuando está sujeto a la recolección de basura. Por lo tanto, un objeto no se destruirá sólo porque el método que lo contiene termine.

HABILIDAD FUNDAMENTAL 6.4

Sobrecarga de métodos

En esta sección, aprenderá acerca de una de las funciones más estimulantes de Java: la sobrecarga de métodos. En Java, dos o más métodos con la misma clase pueden compartir el mismo nombre, siempre y cuando sus declaraciones de parámetros sean diferentes. Cuando sucede así, se dice que los métodos están *sobrecargados*, y al proceso se le denomina *sobrecarga de métodos*. La sobrecarga de métodos es una de las maneras en las que Java implementa el polimorfismo.

En general, para sobrecargar un método, simplemente debe declarar diferentes versiones de él. El compilador se hace cargo del resto. Sin embargo, debe observar una restricción importante: el tipo o el número de parámetros de cada método sobrecargado deben ser diferentes. No basta con que dos métodos difieran sólo en su tipo de regreso. (Los tipos de regreso no proporcionan información suficiente para que Java decida cuál método usar.) Por supuesto, los métodos sobrecargados *pueden* también diferir en sus tipos de regreso. Cuando se llama a un método sobrecargado, se ejecuta la versión del método cuyos parámetros coinciden con el argumento.

He aquí un ejemplo simple que ilustra la sobrecarga de métodos:

```
// Demuestra la sobrecarga de métodos.
class Sobrecarga {
  void soblDemo() {                                   Primera versión
    System.out.println("No hay parámetros");
  }

  // Sobrecarga soblDemo para un parámetro entero.
  void soblDemo(int a) {                              Segunda versión
    System.out.println("Un parámetro: " + a);
  }

  // Sobrecarga soblDemo para dos parámetros enteros.
  int soblDemo(int a, int b) {                        Segunda versión
    System.out.println("Dos parámetros: " + a + " " + b);
    return a + b;
  }

  // Sobrecarga soblDemo para dos parámetros double.
  double soblDemo(double a, double b) {               Cuarta versión
```

```
    System.out.println("Dos parámetros double: " +
                       a + " "+ b);
    return a + b;
  }
}

class SobrecargaDemo {
  public static void main(String args[]) {
    Sobrecarga ob = new Sobrecarga();
    int resI;
    double resD;

    // llama a todas las versiones de soblDemo()
    ob.soblDemo();
    System.out.println();

    ob.soblDemo(2);
    System.out.println();

    resI = ob.soblDemo(4, 6);
    System.out.println("Resultado de ob.soblDemo(4, 6): " +
                       resI);
    System.out.println();


    resD = ob.soblDemo(1.1, 2.32);
    System.out.println("Resultado de ob.soblDemo(1.1, 2.32): " +
                       resD);
  }
}
```

Este programa genera la siguiente salida:

```
No hay parámetros

Un parámetro: 2

Dos parámetros: 4 6
Resultado de ob.soblDemo(4, 6): 10

Dos parámetros double: 1.1 2.32
Resultado de ob.soblDemo(1.1, 2.32): 3.42
```

Como puede ver, **sob1Demo()** está sobrecargado cuatro veces. La primera versión no toma parámetros, la segunda toma un parámetro entero, la tercera toma dos parámetros enteros y la cuarta toma dos parámetros **double**. Observe que las primeras dos versiones de **sob1Demo()** regresan **void**

y las siguientes dos regresan un valor. Esto es perfectamente válido pero, como ya se explicó, la sobrecarga no se ve afectada de una manera u otra por el tipo de regreso de un método. Así pues, si se trata de usar estas dos versiones de **sob1Demo()**, se producirá un error.

```
// Un sob1Demo(int) está OK.
void sob1Demo(int a) {
  System.out.println("Un parámetro " + a);
}
/* ¡Error! Dos sob1Demo no está OK.
    Aunque difieran los tipos de regreso.
*/
int sob1Demo(int a) {
  System.out.println("Un parámetro " + a);
  Return a * a;
}
```

No es posible usar los tipos de regreso para diferenciar métodos sobrecargados.

Como el comentario sugiere, la diferencia en sus tipos de regreso no es suficiente para los objetivos de la sobrecarga.

Como recordará a partir del módulo 2, Java proporciona ciertas conversiones automáticas de tipo. Estas conversiones también aplican a parámetros de métodos sobrecargados. Por ejemplo, considere lo siguiente:

```
/* Las conversiones automáticas de tipo
   afectan la resolución del método sobrecargado.
*/
class Sobrecarga2 {
  void f(int x) {
    System.out.println("Dentro de f(int): " + x);
  }

  void f(double x) {
    System.out.println("Dentro de f(double): " + x);
  }
}

class ConvTipo {
  public static void main(String args[]) {
    Sobrecarga2 ob = new Sobrecarga2();

    int i = 10;
    double d = 10.1;
```

```
    byte b = 99;
    short s = 10;
    float f = 11.5F;


    ob.f(i); // llama a ob.f(int)
    ob.f(d); // llama a ob.f(double)

    ob.f(b); // llama a ob.f(int) -- conversión de tipo
    ob.f(s); // llama a ob.f(int) -- conversión de tipo
    ob.f(f); // llama a ob.f(double) -- conversión de tipo
  }
}
```

Ésta es la salida del programa:

```
Dentro de f(int): 10
Dentro de f(double): 10.1
Dentro de f(int): 99
Dentro de f(int): 10
Dentro de f(double): 11.5
```

En este ejemplo, sólo dos versiones de **f( )** están definidas; una que tiene un parámetro **int** y otra que tiene un parámetro **double**. Sin embargo, es posible pasar un valor **byte**, **short** o **float** a **f( )**. En el caso de **byte** y **short**, Java los convierte automáticamente en **int**, así que se invoca **f(int)**. En el caso de **float( )**, el valor se convierte en **double** y se llama a **f(double)**.

Sin embargo, es importante comprender que la conversión automática sólo aplica si no hay una coincidencia directa entre un parámetro y un argumento. Por ejemplo, he aquí el programa anterior con la adición de una versión de **f( )** que especifica un parámetro **byte**.

```
// Agregar f(byte).
class Sobrecarga2 {
  void f(byte x) {
    System.out.println("Dentro de f(byte): " + x);
  }

  void f(int x) {
    System.out.println("Dentro de f(int): " + x);
  }

  void f(double x) {
    System.out.println("Dentro de f(double): " + x);
```

```
  }
}

class ConvTipo {
  public static void main(String args[]) {
    Sobrecarga2 ob = new Sobrecarga2();

    int i = 10;
    double d = 10.1;

    byte b = 99;
    short s = 10;
    float f = 11.5F;


    ob.f(i); // llama a ob.f(int)
    ob.f(d); // llama a ob.f(double)

    ob.f(b); // llama a  ob.f(byte) -- ahora no hay conversión de tipo

    ob.f(s); // llama a  ob.f(int) -- conversión de tipo
    ob.f(f); // llama a  ob.f(double) -- conversión de tipo
  }
}
```

Ahora, cuando el programa se ejecuta, se produce la siguiente salida:

```
Dentro de f(int): 10
Dentro de f(double): 10.1
Dentro de f(byte): 99
Dentro de f(int): 10
Dentro de f(double): 11.5
```

En esta versión, como hay una versión de **f( )** que toma un argumento **byte**, cuando se llama a **f( )** con un argumento **byte**, se invoca a **f(byte)** y no ocurre la conversión automática a **int**.

La sobrecarga de método soporta el polimorfismo porque es una manera en la que Java implementa el paradigma "una interfaz, varios métodos". Para comprender cómo se realiza esto, piense en lo siguiente: en lenguajes que no soportan la sobrecarga de métodos, debe asignarse a cada método un nombre único. Sin embargo, con frecuencia querrá implementar esencialmente el mismo método para diferentes tipos de datos. Considere la función de valor absoluto: en lenguajes que no soportan

la sobrecarga, suele haber tres o más versiones de esta función, cada una con un nombre ligeramente diferente. Por ejemplo, en C, la función **abs()** regresa el valor absoluto de un entero, la función **labs()** devuelve el valor absoluto de un entero largo, y **fabs()** regresa el valor absoluto de un valor de punto flotante. Como C no soporta la sobrecarga, cada función debe tener su propio nombre, aunque las tres funciones realicen, en esencia, lo mismo. Esto hace que, conceptualmente, la situación sea más compleja de lo que es. Aunque el concepto de cada función es el mismo, es necesario que recuerde tres nombres. Esta situación no se presenta en Java porque cada método de valor absoluto puede usar el mismo nombre. Por supuesto, la biblioteca estándar de clases de Java incluye un método de valor absoluto llamado **abs()**. Este método está sobrecargado por la clase **Math** para manejar los tipos numéricos. Java determina la versión de **abs()** que se llama con base en el tipo de argumento.

El valor de la sobrecarga radica en que permite que reacceder a métodos relacionados por el uso de un nombre común. Por lo tanto, el nombre **abs** representa la *acción general* que se está realizando. El compilador debe elegir la versión *específica* correcta para una circunstancia particular. Usted, el programador, sólo necesita recordar la operación general que se está llevando a cabo. Mediante la aplicación del polimorfismo, varios nombres se han reducido a uno. Aunque este ejemplo es muy simple, si expande el concepto, observará la manera en que la sobrecarga puede ayudar a manejar una mayor complejidad.

Cuando sobrecarga un método, cada versión de dicho método puede realizar cualquier actividad que desee. No hay una regla que establezca que los métodos sobrecargados deban relacionarse entre sí. Sin embargo, desde un punto de vista estilístico, la sobrecarga de un método implica una relación. Por consiguiente, aunque pueda usar el mismo nombre para sobrecargar métodos no relacionados, no debe hacerlo. Por ejemplo, podría usar el nombre **cuad** para crear métodos que regresen *el cuadrado* de un entero y la *raíz cuadrada* de un valor de punto flotante. Sin embargo, estas dos operaciones son fundamentalmente diferentes. Aplicar de esta manera la sobrecarga de métodos va en contra de su propósito original: en la práctica, sólo debe sobrecargar operaciones estrechamente relacionadas.

Pregunte al experto

P: **He oído a los programadores de Java utilizar el término *firma*. ¿A qué se refiere éste?**

R: En lo que se refiere a Java, una firma es el nombre de un método, más su lista de parámetros. Así que, para los fines de la sobrecarga, dos métodos dentro de la misma clase no pueden tener la misma firma. Tome en cuenta que una firma no incluye el tipo de regreso porque Java no lo utiliza para la resolución de sobrecarga.

Comprobación de avance

1. Para que un método se sobrecargue, ¿cuál condición debe cumplirse?
2. ¿El tipo de regreso desempeña una función en la sobrecarga de métodos?
3. ¿De qué manera la conversión automática de tipos de Java afecta a la sobrecarga?

HABILIDAD FUNDAMENTAL 6.5

Sobrecarga de constructores

Al igual que los métodos, los constructores también pueden sobrecargarse. Esto le permite construir objetos de varias maneras. Por ejemplo, considere el siguiente programa:

```
// Demuestra un constructor sobrecargado.
class MiClase {
  int x;

  MiClase() {
    System.out.println("Dentro de MiClase().");
    x = 0;
  }

  MiClase(int i) {
    System.out.println("Dentro de MiClase(int).");
    x = i;
  }

  MiClase(double d) {
    System.out.println("Dentro de MiClase(double).");
    x = (int) d;
  }

  MiClase(int i, int j) {
    System.out.println("Dentro de MiClase(int, int).");
```

Construye objetos de varias maneras.

1. Para que un método sobrecargue a otro, el tipo y/o el número de parámetros deben ser diferentes.
2. No. El tipo de regreso puede diferir entre los métodos sobrecargados, pero no afecta a la sobrecarga de métodos de ninguna manera.
3. Cuando no exista una coincidencia directa entre un conjunto de argumentos y un conjunto de parámetros, el método con el conjunto de parámetros que coincida más estrechamente será el que se utilice si los argumentos pueden convertirse automáticamente al tipo de los parámetros.

```
    x = i * j;
  }
}

class ConsSobrecargaDemo {
  public static void main(String args[]) {
    MiClase t1 = new MiClase();
    MiClase t2 = new MiClase(88);
    MiClase t3 = new MiClase(17.23);
    MiClase t4 = new MiClase(2, 4);

    System.out.println("t1.x: " + t1.x);
    System.out.println("t2.x: " + t2.x);
    System.out.println("t3.x: " + t3.x);
    System.out.println("t4.x: " + t4.x);
  }
}
```

Ésta es la salida del programa:

```
Dentro de MiClase().
Dentro de MiClase(int).
Dentro de MiClase(double).
Dentro de MiClase(int, int).
t1.x: 0
t2.x: 88
t3.x: 17
t4.x: 8
```

MiClase( ) está sobrecargada de cuatro formas, y cada una construye un objeto de manera diferente. Se llama al constructor apropiado con base en los parámetros especificados cuando se ejecuta **new**. Al sobrecargar un constructor de clase, usted brinda al usuario de su clase flexibilidad en la manera en que se construyen los objetos.

Una de las razones más comunes por la que los constructores se sobrecargan es para permitir que un objeto inicialice a otro. Por ejemplo, considere este programa que usa la clase **Sumatoria** para calcular la sumatoria de un valor entero.

```
// Inicializa un objeto con otro.
class Sumatoria {
  int suma;
```

```
  // Construye desde un  int.
  Sumatoria(int num) {
    suma = 0;
    for(int i=1; i <= num; i++)
      suma += i;
  }

  // Construye desde otro objeto.
  Sumatoria(Sumatoria ob) {
    suma = ob.suma;
  }
}

class SumDemo {
  public static void main(String args[]) {
    Sumatoria s1 = new Sumatoria(5);
    Sumatoria s2 = new Sumatoria(s1);

    System.out.println("s1.suma: " + s1.suma);
    System.out.println("s2.suma: " + s2.suma);
  }
}
```

Construye un objeto a partir de otro.

Aquí se muestra la salida:

```
s1.suma: 15
s2.suma: 15
```

A menudo, como lo muestra este ejemplo, una ventaja de proporcionar un constructor que utilice un objeto para inicializar otro es la eficiencia. En este caso, cuando **s2** se construye, no es necesario volver a realizar la sumatoria. Por supuesto, aún en casos en los que la eficiencia no es importante, proporcionar un constructor que lleve a cabo la copia de un objeto suele resultar útil.

Comprobación de avance

1. ¿Un constructor puede tomar un objeto de su propia clase como parámetro?
2. ¿Por qué es posible que usted desee proporcionar constructores sobrecargados?

1. Sí.
2. Para proporcionar conveniencia y flexibilidad al usuario de su clase.

Proyecto 6.2 Sobrecarga del constructor Cola

CDemo2.java

En este proyecto mejorará la clase **Cola** al proporcionar dos constructores adicionales. El primero construirá una nueva cola a partir de otra y el segundo construirá una cola, dándole un valor inicial. Como verá, la adición de estos constructores mejora sustancialmente la utilidad de **Cola**.

Paso a paso

1. Cree un archivo llamado **CDemo2.java** y copie en él la clase actualizada **Cola** del proyecto 6.1.

2. En primer lugar, agregue el siguiente constructor, el cual construye una cola a partir de Cola.

```
// Construye una Cola a partir de Cola.
Cola(Cola ob) {
  colocarlug = ob.colocarlug;
  obtenerlug = ob.obtenerlug;
  q = new char[ob.q.length];

  // copia elementos
  for(int i=obtenerlug+1; i <= colocarlug; i++)
    q[i] = ob.q[i];
}
```

Revisemos de cerca este constructor: inicializa **colocarlug** y **obtenerlug** al valor contenido en el parámetro **ob**; luego asigna una nueva matriz para contener la cola y copia los elementos de **ob** en esta matriz. Una vez construida, la nueva cola será una copia idéntica a la original, pero ambas serán objetos completamente separados.

3. A continuación agregue el constructor que inicializa la cola a partir de una matriz de caracteres, como se muestra aquí:

```
// Construye una cola con valores iniciales.
Cola(char a[]) {
  colocarlug = 0;
  obtenerlug = 0;
  q = new char[a.length+1];

  for(int i = 0; i < a.length; i++) colocar(a[i]);
}
```

Este constructor crea una cola del largo suficiente como para contener los caracteres de **a** y luego los almacena en la cola. Debido a la manera en que el algoritmo de la cola funciona, la longitud de la cola debe ser 1 mayor que la matriz.

(continúa)

4. A continuación se presenta la clase **Cola** completa junto con la clase **CDemo2** que la demuestra:

```
   Proyecto 6.2
   // Una clase mejorada de cola para caracteres.
class Cola {
  private char q[]; // esta matriz contiene la cola
  private int colocarlug, obtenerlug; // los índices colocar y obtener

  // Construye una cola vacía dado su tamaño.
  Cola(int dimen) {
    q = new char[dimen+1]; // asigna memoria a la cola
    colocarlug = obtenerlug = 0;
  }

  // Construye una Cola a partir de Cola.
  Cola(Cola ob) {
    colocarlug = ob.colocarlug;
    obtenerlug = ob.obtenerlug;
    q = new char[ob.q.length];

    // copia elementos
    for(int i=obtenerlug+1; i <= colocarlug; i++)
      q[i] = ob.q[i];
  }

  // Construye una cola con valores iniciales.
  Cola(char a[]) {
    colocarlug = 0;
    obtenerlug = 0;
    q = new char[a.length+1];

    for(int i = 0; i < a.length; i++) colocar(a[i]);
  }

  // coloca un carácter en la cola
  void colocar(char ch) {
    if(colocarlug==q.length-1) {
      System.out.println(" - La cola se ha llenado.");
      return;
    }

    colocarlug++;
    q[colocarlug] = ch;
  }

  // Obtiene un carácter de la cola
```

```
  char obtener() {
    if(obtenerlug == colocarlug) {
      System.out.println(" - La cola se ha vaciado.");
      return (char) 0;
    }

    obtenerlug++;
    return q[obtenerlug];
  }
}

// Demuestra la clase Cola.
class CDemo2 {
  public static void main(String args[]) {
    // construye una cola vacía de 10 elementos
    Cola q1 = new Cola(10);

    char nombre[] = {'J', 'u', 'a', 'n'};
    // construye la cola desde la matriz
    Cola q2 = new Cola(nombre);

    char ch;
    int i;

    // coloca algunos caracteres en q1
    for(i=0; i < 10; i++)
      q1.colocar((char) ('A' + i));

    // construye una cola de otra cola
    Cola q3 = new Cola(q1);

    // Muestra las colas.
    System.out.print("Contenido de q1: ");
    for(i=0; i < 10; i++) {
      ch = q1.obtener();
      System.out.print(ch);
    }

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

    System.out.print("Contenido de q2: ");
    for(i=0; i < 3; i++) {
      ch = q2.obtener();
      System.out.print(ch);
    }
```

(continúa)

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

    System.out.print("Contenido de q3: ");
    for(i=0; i < 10; i++) {
      ch = q3.obtener();
      System.out.print(ch);

    }
  }
}
```

Aquí se muestra la salida del programa:

```
Contenido de q1: ABCDEFGHIJ

Contenido de q2: Juan

Contenido de q3: ABCDEFGHIJ
```

HABILIDAD FUNDAMENTAL 6.6

Recursión

En Java, un método puede llamarse a sí mismo. A este proceso se le llama *recursión*, y se dice que un método que se llama a sí mismo es *recursivo*. En general, la recursión es el proceso de definir algo en términos de sí mismo y es muy parecido a una definición circular. El componente clave de un método recursivo es una instrucción que ejecuta una llamada a sí misma. La recursión es un mecanismo poderoso de control.

El ejemplo clásico de recursión es el cálculo del factorial de un número. El *factorial* de un número *N* es el producto de todos los números de 1 a *N*. Por ejemplo, 3 factorial es 1 x 2 x 3 o 6. El siguiente programa muestra una manera recursiva de calcular el factorial de un número. Para fines de comparación, se incluye también un componente no recursivo.

```
// Un ejemplo simple de recursión.
class Factorial {
  // Ésta es una función recursiva.
  int factR(int n) {
    int result;

    if(n==1) return 1;
    result = factR(n-1) * n;
    return result;
  }

  // Éste es un equivalente iterativo.
  int factI(int n) {
    int t, result;
```

Ejecuta la llamada recursiva a **factR()**.

```
    result = 1;
    for(t=1; t <= n; t++) result *= t;
    return result;
}
}

class Recursion {
  public static void main(String args[]) {
    Factorial f = new Factorial();

    System.out.println("Factoriales usando método recursivo.");
    System.out.println("El factorial de 3 es " + f.factR(3));
    System.out.println("El factorial de 4 es " + f.factR(4));
    System.out.println("El factorial de 5 es " + f.factR(5));
    System.out.println();

    System.out.println("Factoriales usando método iterativo.");
    System.out.println("El factorial de 3 es " + f.factI(3));
    System.out.println("El factorial de 4 es " + f.factI(4));
    System.out.println("El factorial de 5 es " + f.factI(5));
  }
}
```

Ésta es la salida de este programa:

```
Factoriales mediante un método recursivo.
El factorial de 3 es 6
El factorial de 4 es 24
El factorial de 5 es 120

Factoriales mediante un método iterativo.
El factorial de 3 es 6
El factorial de 4 es 24
El factorial de 5 es 120
```

La operación del método no recursivo **factI()** debe ser clara. Éste emplea un bucle que empieza en 1 y multiplica de manera progresiva cada número por el producto en progreso.

La operación de **factR()** no recursivo es un poco más compleja. Cuando se llama a **factR()** con un argumento de 1, el método regresa 1; de otra manera regresa el producto de **factR(n-1)*n**. Para evaluar esta expresión, se llama a **factR()** con **n-1**. Este proceso se repite hasta que **n** es igual a 1 y las llamadas empiezan a regresar al método. Por ejemplo, cuando se calcula el factorial de 2, la primera llamada a **factR()** causará una segunda llamada con un argumento de 1. Esta llamada devolverá 1, que luego se multiplicará por 2 (el valor original de **n**). La respuesta es entonces 2. Tal vez le resulte interesante insertar instrucciones **println()** en **factR()** que muestren a qué nivel se encuentra cada llamada y cuáles son los resultados intermedios.

Cuando un método se llama a sí mismo, se asigna almacenamiento a nuevas variables locales y parámetros en la pila. Además, el código del método se ejecuta con tres nuevas variables desde el principio. Una llamada recursiva no realiza una nueva copia del método. Sólo los argumentos son nuevos. A medida que regresa cada llamada recursiva, la variable local y los parámetros antiguos se eliminan de la pila y la ejecución se reanuda en el punto de la llamada dentro del método.

Las versiones recursivas de varias rutinas pueden ejecutarse un poco más lentamente que los equivalentes iterativos debido a la sobrecarga que se agrega a las llamadas adicionales al método. Demasiadas llamadas recursivas a un método podrían causar el sobrepaso de una pila. Debido a que el almacenamiento para parámetros y variables locales se encuentra en la pila y cada nueva llamada crea una nueva copia de tres variables, es posible que la pila llegue a agotarse. Si esto ocurre, el sistema en tiempo de ejecución de Java originará una excepción. Sin embargo, tal vez no tenga que preocuparse de esto a menos que una rutina recursiva se salga de control.

La principal ventaja de la recursión es que algunos tipos de algoritmos pueden implementarse de manera más clara y simple de modo recursivo que iterativo. Por ejemplo, el algoritmo de ordenamiento rápido es muy difícil de implementar de manera iterativa. Además, algunos problemas, sobre todo los relacionados con la inteligencia artificial, parecen prestarse mejor a soluciones recursivas.

Cuando escriba métodos recursivos, debe contar en algún lugar con una instrucción condicional, como un **if**, para hacer que el método regrese sin que la llamada recursiva se ejecute. Si no lo hace, una vez que llame al método, nunca regresará. Este tipo de error es muy común cuando se trabaja con recursión. Use instrucciones **println( )** de manera libre para que pueda observar lo que pasa y aborte la ejecución si comete un error.

HABILIDAD FUNDAMENTAL 6.7

Comprensión de static

Habrá ocasiones en las que quiera definir un miembro de clase que se emplee de manera independiente a cualquier objeto de dicha clase. Por lo general, un miembro de clase debe accederse a través de un objeto de su clase; sin embargo, es posible crear un miembro que pueda utilizarse por sí solo, sin referencia a una instancia específica. Para crear este tipo de miembro, anteceda su declaración con la palabra clave **static**. Cuando se declare un miembro **static**, éste puede ser accedido antes de que cualquier objeto de su clase se cree, y sin referencia a ningún objeto. Es posible declarar métodos y variables como **static**. El ejemplo más común de miembro **static** es **main( )**. **main( )** se declara como **static** porque debe ser llamado por el sistema operativo cuando su programa se inicia.

Fuera de la clase, para usar un miembro **static** sólo necesita especificar el nombre de la clase de éste seguido del operador de punto. No es necesario crear ningún objeto. Por ejemplo, si quiere asignar el valor 10 a una variable **static** llamada **cuenta**, que es parte de la clase **Crono**, use esta línea:

```
Crono.cuenta = 10;
```

El formato es similar al usado para acceder a variables de instancia normales mediante un objeto, excepto que se usa el nombre de la clase. Puede llamarse a un método **static** de la misma manera: usando el operador de punto en el nombre de la clase.

Las variables declaradas como **static** son, en esencia, variables globales. Cuando se declara un objeto, no se realiza una copia de **static**. En cambio, todas las instancias de la clase comparten la misma variable **static**. He aquí un ejemplo que muestra la diferencia entre una variable **static** y una de instancia:

```
// Uso de una variable static.
class StaticDemo {
  int x; // una variable de instancia normal
  static int y; // una variable static
}

class SDemo {
  public static void main(String args[]) {
    StaticDemo ob1 = new StaticDemo();
    StaticDemo ob2 = new StaticDemo();

    /* Cada objeto tiene su propia copia de
       una variable de instancia. */
    ob1.x = 10;
    ob2.x = 20;
    System.out.println("Por supuesto, ob1.x y ob2.x " +
                       "son independientes.");
    System.out.println("ob1.x: " + ob1.x +
                       "\nob2.x: " + ob2.x);
    System.out.println();

    /* Cada objeto comparte una copia de
       una variable static. */
    System.out.println("Se comparte la variable y.");
    ob1.y = 19;
    System.out.println("ob1.y: " + ob1.y +
                       "\nob2.y: " + ob2.y);
    System.out.println();

    System.out.println("Se accesa la variable y" +
                       " mediante su clase.");
    StaticDemo.y = 11; // Puede hacer ref a y mediante el nombre de clase
    System.out.println("StaticDemo.y: " + StaticDemo.y +
                       "\nob1.y: " + ob1.y +
                       "\nob2.y: " + ob2.y);
  }
}
```

Hay una copia de y para que todos los objetos la compartan.

A continuación se muestra la salida del programa:

```
Por supuesto, ob1.x y ob2.x son independientes.
ob1.x: 10
ob2.x: 20

Se comparte la variable y.
ob1.y: 19
ob2.y: 19

Se accede a la variable y mediante su clase.
StaticDemo.y: 11
ob1.y: 11
ob2.y: 11
```

Como puede ver, la variable **static** y es compartida por **ob1** y **ob2**. Un cambio a través de una instancia implica un cambio en todas las instancias. Más aún, y puede ser accedido a través de un nombre de objeto, como **ob2.y**, o a través de su nombre de clase, como en **StaticDemo.y.**

La diferencia entre un método **static** y un método normal es que el **static** puede llamarse mediante su nombre de clase, sin que ningún objeto de dicha clase se cree. Ya ha visto un ejemplo de esto: el método **sqrt()**, que es un método **static** dentro de la clase **Math**. A continuación se muestra un ejemplo que crea un método **static**:

```
// Uso de un método static.
class StaticMet {
  static int val = 1024; // una variable static

  // un método static
  static int valDiv2() {
    return val/2;
  }
}

class SDemo2 {
  public static void main(String args[]) {

    System.out.println("val es " + StaticMet.val);
    System.out.println("StaticMet.valDiv2(): " +
                       StaticMet.valDiv2());

    StaticMet.val = 4;
    System.out.println("val es " + StaticMet.val);
    System.out.println("StaticMet.valDiv2(): " +
                       StaticMet.valDiv2());
```

```
  }
}
```

Aquí se muestra la salida:

```
val es 1024
StaticMet.valDiv2(): 512
val es 4
StaticMet.valDiv2(): 2
```

Los métodos declarados como **static** tienen varias restricciones:

- Sólo pueden llamar a otros métodos **static**.
- Sólo deben accederse datos **static**.
- No tienen una referencia **this**.

Por ejemplo, en la siguiente clase, el método static **ValDivDenom()** es ilegal.

```
class StaticError {
  int denom = 3; // una variable de instancia normal
  static int val = 1024; // una variable static

  /* ¡Error! No puede accesar una variable no static
     desde un método static. */
  static int valDivDenom() {
    return val/denom; // no se compila!
  }
}
```

En este caso, **denom** es una variable de instancia normal que no puede ser accedida dentro de un método **static**.

Bloques static

A veces una clase requerirá algún tipo de inicialización antes de que esté lista para crear objetos. Por ejemplo, tal vez se requiera establecer una conexión con un sitio remoto. Quizás también sea necesario inicializar ciertas variables **static** antes de que se emplee cualquier de los métodos **static** de la clase. Para manejar este tipo de situaciones Java le permite declarar un bloque **static**. Un bloque **static** se ejecuta cuando la clase se carga por primera vez, por lo que se ejecuta antes de que la clase pueda emplearse para cualquier otro fin. He aquí un ejemplo de un bloque **static**:

```
// Uso de un bloque static
class StaticBloque {
  static double raízde2;
  static double raízde3;
```

```
  static {
    System.out.println("Dentro del bloque static.");
    raízde2 = Math.sqrt(2.0);
    raízde3 = Math.sqrt(3.0);
  }

  StaticBloque(String msj) {
    System.out.println(msj);
  }
}

class SDemo3 {
  public static void main(String args[]) {
    StaticBloque ob = new StaticBloque("Dentro del constructor");

    System.out.println("La raíz cuadrada de 2 es " +
                       StaticBloque.raízde2);
    System.out.println("La raíz cuadrada de 3 es " +
                       StaticBloque.raízde3);

  }
}
```

Este bloque se ejecuta cuando la clase es cargada.

Aquí se muestra la salida:

```
Dentro del bloque static.
Dentro del constructor
La raíz cuadrada de 2 es 1.4142135623730951
La raíz cuadrada de 3 es 1.7320508075688772
```

Como puede observar, el bloque **static** se ejecuta antes de que se construya cualquier objeto.

Comprobación de avance

1. Defina recursión.

2. Explique la diferencia entre variables **static** y variables de instancia.

3. ¿Cuándo se ejecuta un bloque **static**?

1. Recursión es el proceso en el que un método se llama a sí mismo.
2. Cada objeto de una clase cuenta con su propia copia de las variables de instancia definidas por la clase. Cada objeto de una clase comparte una copia de una variable **static**.
3. Un bloque **static** se ejecuta cuando su clase se carga por primera vez, antes de su primer uso.

Proyecto 6.3 El ordenamiento rápido

ORDemo.java

En el módulo 5 se mostró un método simple de ordenamiento llamado de Burbuja. Se mencionó que existen mejores métodos de ordenamiento. A continuación desarrollará una de las mejores versiones: el ordenamiento rápido. Lo inventó C. A. R. Hoare, quien también le dio su nombre (Quicksort, en inglés). Es, además, el mejor algoritmo de ordenamiento de propósito general del que se dispone. La razón de que haya podido mostrarse en el módulo 5 es que la mejor implementación del ordenamiento rápido depende de la recursión. La versión que desarrollaremos ordena una matriz de caracteres, pero la lógica puede adaptarse para ordenar cualquier tipo de objeto que desee.

El ordenamiento rápido está construido sobre la idea de las particiones. El procedimiento general consiste en seleccionar un valor, llamado el *comparando*, y luego dividir la matriz en dos secciones. Todos los elementos mayores o iguales al valor de partición se colocan en un lado, y los menores al valor en el otro. Este proceso se repite entonces para cada sección restante hasta que se ordena la matriz. Por ejemplo, dada la matriz **fedacb** y usando el valor **d** como comparando, el primer paso del ordenamiento rápido reorganizaría la matriz de la siguiente manera:

Inicial	f e d a c b
Paso 1	b c a d e f

Posteriormente, este proceso se repite para cada sección (es decir, **bca** y **def**). Como verá, el proceso es de naturaleza esencialmente recursiva y, por supuesto, la implementación más limpia del ordenamiento rápido es recursiva.

Es posible seleccionar el valor del comparando de dos maneras: puede elegirlo al azar o seleccionarlo al promediar un pequeño conjunto de valores tomados de la matriz. Para un ordenamiento óptimo, debe seleccionar un valor que se encuentre precisamente en medio del rango de valores. Sin embargo, esto no resulta fácil de llevar a cabo en casi ningún conjunto de datos. En el peor de los casos, el valor elegido se puede encontrar en un extremo. Sin embargo, aun en este caso, el ordenamiento rápido se realizará correctamente. La versión de ordenamiento rápido que desarrollaremos selecciona como comparando el elemento que se encuentra en medio de la matriz.

Paso a paso

1. Cree un archivo llamado **ORDemo.java**.
2. Primero, cree la clase **OrdenRap** que se muestra aquí:

```
   Proyecto 6.3: Una versión simple de ordenamiento rápido.
class OrdenRap {

  // Configura una llamada al método OrdenRap real.
  static void orapid(char elems[]) {
    or(elems, 0, elems.length-1);
  }
```

(continúa)

```
  // Una versión recursiva de OrdenRap para caracteres.
  private static void or(char elems[], int izquierda, int derecha)
  {
    int i, j;
    char x, y;

    i = izquierda; j = derecha;
    x = elems[(izquierda+derecha)/2];

    do {
      while((elems[i] < x) && (i < derecha)) i++;
      while((x < elems[j]) && (j > izquierda)) j--;

      if(i <= j) {
        y = elems[i];
        elems[i] = elems[j];
        elems[j] = y;
        i++; j--;
      }
    } while(i <= j);

    if(izquierda < j) or(elems, izquierda, j);
    if(i < derecha) or(elems, i, derecha);
  }
}
```

Para mantener la simplicidad de la interfaz de **OrdenRap**, la clase **OrdenRap** proporciona el método **orapid( )** que configura una llamada al método de ordenamiento rápido real, **or( )**. Esto permite que se llame al ordenamiento rápido con el sólo nombre de la matriz que habrá de ordenarse, sin tener que proporcionar una partición inicial. Debido a que **or( )**se usa sólo internamente, se especifica como **private**.

3. Para usar **OrdenRap**, simplemente llame a **OrdenRap.orapid( )**. Debido a que **orapid( )** se especifica como **static**, puede llamarse a través de su clase en lugar de hacerlo como un objeto. Por lo tanto, no hay necesidad de crear un objeto **OrdenRap**. Después de que regrese la llamada, la matriz se ordenará. Recuerde que esta versión sólo funciona con matrices de caracteres, pero puede adaptar la lógica para ordenar cualquier tipo de matriz que desee.

4. He aquí el programa que demuestra **OrdenRap**:

```
   Proyecto 6.3: Una versión simple de ordenamiento rápido.
class OrdenRap {

  // Configura una llamada al método OrdenRap real.
  static void orapid(char elems[]) {
    or(elems, 0, elems.length-1);
  }
```

```
  // Una versión recursiva de OrdenRap para caracteres.
  private static void or(char elems[], int izquierda, int derecha)
  {
    int i, j;
    char x, y;

    i = izquierda; j = derecha;
    x = elems[(izquierda+derecha)/2];

    do {
      while((elems[i] < x) && (i < derecha)) i++;
      while((x < elems[j]) && (j > izquierda)) j--;

      if(i <= j) {
        y = elems[i];
        elems[i] = elems[j];
        elems[j] = y;
        i++; j--;
      }
    } while(i <= j);

    if(izquierda < j) or(elems, izquierda, j);
    if(i < derecha) or(elems, i, derecha);
  }
}

class ORDemo {
  public static void main(String args[]) {
    char a[] = { 'd', 'x', 'a', 'r', 'p', 'j', 'i' };
    int i;

    System.out.print("Matriz original: ");
    for(i=0; i < a.length; i++)
      System.out.print(a[i]);

    System.out.println();

    // ahora, se ordena la matriz
    OrdenRap.orapid(a);

    System.out.print("Matriz ordenada: ");
    for(i=0; i < a.length; i++)
      System.out.print(a[i]);
  }
}
```

HABILIDAD FUNDAMENTAL 6.8

Introducción a clases anidadas e internas

En Java puede definir una *clase anidada*. Se trata de una clase que se declara dentro de otra clase. Francamente, las clases anidadas son un tema avanzado. En realidad, las clases anidadas ni siquiera se permitían en la primera versión de Java. Fue hasta Java 1.1 que se agregaron. Sin embargo, es importante que sepa lo que son y su mecánica de uso pues desempeñan una función importante en muchos programas reales.

Una clase anidada sólo es conocida para la clase que la contiene. Por lo tanto, el alcance de la clase anidada está limitado por la de su clase externa. Una clase anidada tiene acceso a los miembros, incluyendo a los miembros privados, de la clase en que está anidada. Sin embargo, la clase que la incluye no tiene acceso a los miembros de la clase anidada.

Hay dos tipos generales de clases anidadas: las precedidas por el modificador **static** y las que no lo están. En este libro nos ocuparemos únicamente de la variedad no estática. A este tipo de clase anidada también se le denomina *clase interna*. Tiene acceso a todas las variables y los métodos de su clase externa y puede hacer referencia a ellos directamente de la misma manera en la que otros miembros no **static** de la clase externa lo hacen.

En ocasiones una clase interna sirve para proporcionar un conjunto de servicios que sólo se usa en la clase que la incluye. He aquí un ejemplo que utiliza una clase interna para calcular varios valores para la clase que la contiene:

```
// Uso de una clase interna.
class Exterior {
  int nums[];

  Exterior(int n[]) {
    nums = n;
  }

  void Analiza() {
    Interior inOb = new Interior();

    System.out.println("Mínimo: " + inOb.min());
    System.out.println("Máximo: " + inOb.max());
    System.out.println("Promedio: " + inOb.prom());
  }

  // Ésta es una clase interna.
  class Interior {
    int min() {
      int m = nums[0];
```

Clase interna

```
      for(int i=1; i < nums.length; i++)
        if(nums[i] < m) m = nums[i];

      return m;
    }

    int max() {
      int m = nums[0];
      for(int i=1; i < nums.length; i++)
        if(nums[i] > m) m = nums[i];

      return m;
    }

    int prom() {
      int a = 0;
      for(int i=0; i < nums.length; i++)
        a += nums[i];

      return a / nums.length;
    }
  }
}

class ClaseAnidadaDemo {
  public static void main(String args[]) {
    int x[] = { 3, 2, 1, 5, 6, 9, 7, 8 };
    Exterior extOb = new Exterior(x);

    extOb.Analiza();
  }
}
```

Aquí se muestra la salida de este programa:

```
Mínimo: 1
Máximo: 9
Promedio: 5
```

En este ejemplo, la clase interna **Interior** calcula varios valores de la matriz **nums**, la cual es un miembro de **Exterior**. Como ya se explicó, una clase anidada tiene acceso a los miembros de la clase que la contiene, de modo que para **Interior** resulta perfectamente aceptable acceder directamente a la matriz **nums**. Claro está que la situación opuesta no es verdadera. Por ejemplo, no es posible que **Analiza()** invoque directamente el método **min()** sin que cree un objeto **Interior**.

Es posible anidar una clase dentro de cualquier alcance de bloque. Al hacerlo, simplemente se crea una clase localizada que no es conocida fuera del bloque. El siguiente ejemplo adapta la clase **MostrarBits** que se desarrolló en el proyecto 5.3 para utilizarla como clase local.

```
// Uso de MostrarBits como clase local.
class ClaseLocalDemo {
  public static void main(String args[]) {

    // Una versión de clase interior de MostrarBits.
    class MostrarBits {
      int numbits;

      MostrarBits(int n) {
        numbits = n;
      }

      void mostrar(long val) {
        long masc = 1;

        // desplaza a la izquierda un 1 a la posición apropiada
        masc <<= numbits-1;

        int espaciador = 0;
        for(; masc != 0; masc >>>= 1) {
          if((val & masc) != 0) System.out.print("1");
          else System.out.print("0");
          espaciador++;
          if((espaciador % 8) == 0) {
            System.out.print(" ");
            espaciador = 0;
          }
        }
        System.out.println();
      }
    }


    for(byte b = 0; b < 10; b++) {
      MostrarBits evalbyte = new MostrarBits(8);

      System.out.print(b + " en binario: ");
      evalbyte.mostrar(b);
    }
  }
}
```

Una clase local anidada dentro de un método.

La salida de esta versión del programa se muestra a continuación:

```
0 en binario: 00000000
1 en binario: 00000001
2 en binario: 00000010
3 en binario: 00000011
4 en binario: 00000100
5 en binario: 00000101
6 en binario: 00000110
7 en binario: 00000111
8 en binario: 00001000
9 en binario: 00001001
```

En este ejemplo, la clase **MostrarBits** no es conocida fuera de **main( )** por lo que cualquier intento de accederla mediante cualquier otro método diferente a **main( )** arrojará como resultado un error.

Un último tema: puede crear una clase interna que no tenga nombre. A esto se le denomina *clase interna anónima*. Un objeto de una clase interna anónima se inicia cuando la clase se declara usando **new**.

Pregunte al experto

P: **¿Cuál es la diferencia entre una clase anidada** static **y una no** static**?**

R: Una clase anidada **static** es aquella que tiene aplicado el modificador **static**. Debido a que es estática, sólo puede acceder directamente a otros miembros **static** de la clase que la contiene. Para acceder a otros miembros de su clase externa debe utilizar una referencia a objeto.

Comprobación de avance

1. Una clase anidada tiene acceso a los demás miembros de la clase que la contiene. ¿Cierto o falso?
2. Una clase anidada es conocida fuera de la clase que la contiene. ¿Cierto o falso?

1. Cierto.
2. Falso.

HABILIDAD FUNDAMENTAL 1.1

Varargs: argumentos de longitud variable

Es probable que en ocasiones desee crear un método que tome un número variable de argumentos con base en su uso preciso. Por ejemplo, un método que abra una conexión de Internet podría tomar el nombre de usuario, la contraseña, el nombre de archivo, el protocolo, etc., pero proporcionar opciones predeterminadas si no se proporcionara parte de esta información. En esta situación, sería conveniente pasar sólo los argumentos para las cuales no se aplican las opciones predeterminadas. Para crear este tipo de método se requiere que alguna forma de crear una lista de argumentos que tenga una longitud variable en lugar de una fija.

En el pasado, los métodos que requerían una lista de argumentos de longitud variable se manejaban de dos maneras; sin embargo, ninguna de las dos resultaba particularmente satisfactoria. En primer lugar, si el número máximo de argumentos era pequeño y conocido, entonces se creaban versiones sobrecargadas del método, una para cada forma en la que el método podía ser llamado. Aunque esto funciona, resulta adecuado y sólo se aplica a una clase particular de situaciones. En casos donde el número máximo de argumentos posibles es más largo o desconocido, se empleaba un segundo método en el que los argumentos se ponían en una matriz, y ésta se pasaba al método. Francamente, ambos métodos solían proporcionar soluciones complejas, de manera que era ampliamente sabido que se necesitaba un método mejor para manejar listas de argumentos de longitud variable.

J2SE 5 agregó una nueva función a Java que simplificó la creación de métodos que requerían tomar un número variable de argumentos. A esta función se le conoce como *varargs*. A un método que toma una cantidad variable de argumentos se le denomina *metódo de argumentos variables* o simplemente *método varargs*. La lista de parámetros para un método varargs no es de longitud fija, sino variable. Por lo tanto, un método varargs puede tomar una cantidad variable de argumentos.

Fundamentos de varargs

Un argumento de longitud variable se especifica con tres puntos **(...)**. Por ejemplo, a continuación se describe la manera de escribir un método llamado **vaPrueba()** que toma un número variable de argumentos:

```
// vaPrueba() usa vararg.
static void vaPrueba(int ... v) {
  System.out.println("Cantidad de args: " + v.length);
  System.out.println("Contenido: ");

  for(int i=0; i < v.length; i++)
    System.out.println("  arg " + i + ": " + v[i]);

  System.out.println();

}
```

Declara una lista de argumentos de longitud variable.

Observe que **v** está declarada como se muestra aquí:

```
int ... v
```

Esta sintaxis le indica al compilador que **vaPrueba( )** puede llamarse con cero o más argumentos. Más aún, hace que **v** se declare implícitamente como una matriz de tipo **int[ ]**. Por consiguiente, dentro de **vaPrueba( )**, es posible acceder a **v** empleando la sintaxis normal de la matriz.

He aquí un programa completo que demuestra **vaPrueba( )**:

```
// Demuestra argumentos de longitud variable.
class VarArgs {

  // vaPrueba() usa vararg.
  static void vaPrueba(int ... v)
    System.out.println("Cantidad de args: " + v.length);
    System.out.println("Contenido: ");

    for(int i=0; i < v.length; i++)
      System.out.println("  arg " + i + ": " + v[i]);

    System.out.println();
  }

  public static void main(String args[])
  {

    // Observe cómo puede llamarse a vaPrueba()
    // con una cantidad variable de argumentos.
    vaPrueba(10);      // 1 arg
    vaPrueba(1, 2, 3); // 3 args
    vaPrueba();        // no args
  }
}
```

Llamada con diferentes cantidades de argumentos.

Ésta es la salida del programa:

```
Cantidad de args: 1
Contenido:
  arg 0: 10

Cantidad de args: 3
Contenido:
  arg 0: 1
  arg 1: 2
  arg 2: 3

Cantidad de args: 0
Contenido:
```

Note dos elementos importantes acerca de este programa. En primer lugar, como ya se explicó, dentro de **vaPrueba( )**, **v** se opera como una matriz. Esto es porque ***v*** *es una matriz*. La sintaxis ... simplemente le indica al compilador que se usará una cantidad variable de argumentos, y que esos argumentos se almacenarán en la matriz a la que **v** hace referencia. En segundo lugar, en **main( )**, **vaPrueba( )** es llamado con cantidades diferentes de argumentos, incluyendo la falta de argumentos. Los argumentos son colocados automáticamente en una matriz y pasados a **v**. En el caso de la falta de argumentos, la longitud de la matriz es cero.

Un método puede tener parámetros "normales" junto con un parámetro de longitud variable. Sin embargo, el parámetro de longitud variable debe ser el último declarado por el método. Por ejemplo, esta declaración de método es perfectamente aceptable:

```
int hazlo(int a, int b, double c, int ... vals) }
```

En este caso, los primeros tres argumentos utilizados en una llamada a **hazlo( )** coinciden con los primeros tres parámetros. Luego, se infiere que cualquier argumento restante pertenece a **vals**.

He aquí la versión retrabajada del método **vaPrueba( )** que toma un argumento regular y uno de longitud variable:

```
// Uso de varargs con argumentos estándar.
class VarArgs2 {

  // Aquí, msj es un parámetro normal y v es
  // un parámetro de varargs.
  static void vaPrueba(String msj, int ... v) {
    System.out.println(msj + v.length);
    System.out.println("Contenido: ");

    for(int i=0; i < v.length; i++)
      System.out.println("  arg " + i + ": " + v[i]);

    System.out.println();
  }

  public static void main(String args[])
  {
    vaPrueba("Un vararg: ", 10);
    vaPrueba("Tres varargs: ", 1, 2, 3);
    vaPrueba("Sin varargs: ");
  }

}
```

Un parámetro "normal" y uno varargs.

Ésta es la salida de este programa:

```
Un vararg: 1
Contenido:
```

```
  arg 0: 10

Tres varargs: 3
Contenido:
  arg 0: 1
  arg 1: 2
  arg 2: 3

Sin varargs: 0
Contenido:
```

Recuerde que el parámetro de varargs debe ser el último. Por ejemplo, la siguiente declaración es incorrecta:

```
int hazlo(int a, int b, double c, int ... vals, boolean dejadeMarcar) { //
¡Error!
```

En este caso, hay un intento de declarar un parámetro regular después del parámetro varargs, lo que resulta ilegal.

Es necesario tomar en cuenta una restricción más: sólo debe haber un parámetro varargs. Por ejemplo, esta declaración tampoco es válida:

```
int hazlo(int a, int b, double c, int ... vals, double ... masvals) { //
¡Error!
```

El intento de declarar el segundo parámetro varargs es ilegal.

Comprobación de avance

1. Muestre cómo declarar un método llamado **suma()** que tome un número variable de argumentos **int**. (Use un tipo de regreso int).
2. Dada esta declaración:

   ```
   void m(double ... x)
   ```

   el parámetro **x** está declarado implícitamente como _____.

1. int suma(int ... n)
2. una matriz de **double**.

Sobrecarga de métodos varargs

Puede sobrecargar un método que tome un argumento de longitud variable. Por ejemplo, el siguiente programa sobrecarga tres veces **vaPueba()**:

```
// Varargs y sobrecarga.
class VarArgs3 {

  static void vaPrueba(int ... v) {
    System.out.println("vaPrueba(int ...): " +
                       "Cantidad de args: " + v.length);
    System.out.println("Contenido: ");

    for(int i=0; i < v.length; i++)
      System.out.println("  arg " + i + ": " + v[i]);

    System.out.println();
  }

  static void vaPrueba(boolean ... v) {
    System.out.println("vaPrueba(boolean ...): " +
                       "Cantidad de args: " + v.length);
    System.out.println("Contenido: ");

    for(int i=0; i < v.length; i++)
      System.out.println("  arg " + i + ": " + v[i]);

    System.out.println();
  }

  static void vaPrueba(String msj, int ... v) {
    System.out.println("vaPrueba(String, int ...): " +
                       msj + v.length);
    System.out.println("Contenido: ");

    for(int i=0; i < v.length; i++)
      System.out.println("  arg " + i + ": " + v[i]);

    System.out.println();
  }

  public static void main(String args[])
  {
    vaPrueba(1, 2, 3);
    vaPrueba("Probando: ", 10, 20);
    vaPrueba(true, false, false);
  }
}
```

Primera versión de **vaPrueba()**.

Segunda versión de **vaPrueba()**.

Tercera versión de **vaPrueba()**.

La salida producida por este programa se muestra a continuación:

```
vaPrueba(int ...): Cantidad de args: 3
Contenido:
  arg 0: 1
  arg 1: 2
  arg 2: 3

vaPrueba(String, int ...): Probando: 2
Contenido:
  arg 0: 10
  arg 1: 20

vaPrueba(boolean ...): Cantidad de args: 3
Contenido:
  arg 0: true
  arg 1: false
  arg 2: false
```

Este programa ilustra las dos maneras en las que un método varargs puede sobrecargarse. En primer lugar, los tipos del parámetro varargs del método pueden diferir. Es el caso de **vaPrueva(int ...)** y **vaPrueba(boolean ...)**. Recuerde que los ... hacen que el parámetro sea tratado como una matriz del tipo especificado. Por lo tanto, así como puede sobrecargar métodos empleando diferentes tipos de parámetros de matriz, puede sobrecargar métodos varargs empleando diferentes tipos de varargs. En este caso, Java usa la diferencia de tipo para determinar a cuál método sobrecargado llamar.

La segunda manera de sobrecargar un método varargs consiste en agregar un parámetro normal. Esto es lo que se hace con **vaPrueba(String, int ...)**. En este caso, Java emplea el número de argumentos y el tipo de éstos para determinar a cuál método llamar.

Varargs y ambigüedad

Es posible que se obtengan errores no esperados cuando se sobrecarga un método que toma un argumento de longitud variable. Estos errores incluyen la ambigüedad, ya que es posible crear una llamada ambigua a un método varargs sobrecargado. Por ejemplo, considere el siguiente programa:

```
// Varargs, sobrecarga y ambigüedad.
//
// ¡Este programa contiene un error
// y no se compila!
class VarArgs4 {

  // Uso de un parámetro vararg int.
  static void vaPrueba(int ... v) {   <-- Una varargs int.
    // ...
  }
```

```
  // Uso de un parámetro vararg boolean.
  static void vaPrueba(boolean ... v) {   <-- Una varargs
    // ...
  }


  public static void main(String args[])
  {
    vaPrueba(1, 2, 3);  // OK
    vaPrueba(true, false, false); // OK

    vaPrueba(); // ¡Error: Ambiguo!   <-- ¡Ambiguo!
  }
}
```

En este programa, la sobrecarga de **vaPrueba()** resulta perfectamente correcta. Sin embargo, este programa no se compilará debido a la siguiente llamada:

```
vaPrueba(); // ¡Error: Ambiguo!
```

Debido a que es posible que el parámetro varargs esté vacío, esta llamada puede traducirse a una **vaPrueva(int ...) o vaPrueba(boolean ...)**. Ambas son igualmente válidas. Por consiguiente, la llamada es inherentemente ambigua.

He aquí otro ejemplo de ambigüedad. Las siguientes versiones sobrecargadas de **vaPrueba** son inherentemente ambiguas a pesar de que una toma un parámetro normal:

```
static void vaPrueba(int ... v) { // ...
static void vaPrueba(int n, int ... v) { // ...
```

Aunque las listas de parámetros de **vaPrueba()** sean diferentes, no hay manera de que el compilador resuelva la siguiente llamada.

vaPrueba(1)

¿Esto se traduce en una llamada a **vaPrueba(int ...)** con un argumento varargs, o en una llamada a **vaPrueba(int, int ...)** sin argumentos? No es posible que el compilador responda esta pregunta, así que la situación es ambigua.

Debido a los errores de ambigüedad como los mostrados, en ocasiones necesitará omitir la sobrecarga y simplemente usar dos nombres diferentes de método. Además, en algunos casos, los errores de ambigüedad exponen una falla conceptual en su código, la cual puede remediar si trabaja cuidadosamente en una solución.

Comprobación de dominio del módulo 6

1. Dado este fragmento,

```
class X  {
  private int cuenta;
```

¿el siguiente programa es correcto?

```
class  Y {
  public static void main(String args[]) {
    X ob = new X();

    ob.cuenta = 10;
```

2. Un especificador de acceso debe _____ una declaración de miembro.

3. El complemento de una cola es una pila: utiliza el acceso primero en entrar, último en salir y suele compararse con una pila de platos. El primer plato puesto en la mesa es el último en usarse. Cree una clase de pila llamada **Pila** que contenga caracteres; llame a los métodos que acceden a la pila **quitar()** y **poner()**; permita al usuario especificar el tamaño de la pila cuando ésta se cree, y mantenga como private a todos los demás miembros de la clase **Pila**. (Sugerencia: puede usar la clase **Cola** como modelo; sólo cambie la manera en la que se tiene acceso a los datos.)

4. Dada esta clase:

```
class Prueba  {
int a;
Prueba(int a) { a = i; }
}
```

escriba un método llamado **cambiar()** que intercambie el contenido de los objetos a que hacen referencia las dos referencias a objeto de **Prueba**.

5. ¿El siguiente fragmento es correcto?

```
class X  {
  int met(int a, int b) { ... }
  String met(int a, int b) { ... }
```

6. Escriba un método recursivo que despliegue hacia atrás el contenido de una cadena.

7. Si todos los objetos de una clase necesitan compartir la misma variable, ¿cómo debe declarar esa variable?

8. ¿Por qué necesitaría usar un bloque **static**?

9. ¿Qué es una clase interna?

10. Para que un miembro sea accesible únicamente por otro miembro de su clase, ¿qué especificador de acceso debe utilizarse?

11. El nombre de un método, más su lista de parámetros, constituye _____ del método.

12. Un argumento **int** se pasa a un método usando una llamada por _____.

13. Cree un método varargs llamado **suma()** que sume los valores **int** que se le pasen. Haga que regrese el resultado. Demuestre su uso.

14. ¿Puede sobrecargarse un método varargs?

15. Muestre un ejemplo de un método varargs sobrecargado que sea ambiguo.

Módulo 7

Herencia

HABILIDADES FUNDAMENTALES

La herencia es uno de los tres principios fundamentales de la programación orientada a objetos porque permite la creación de clasificaciones jerárquicas. Con el uso de la herencia, puede crear una clase general que defina rasgos comunes a un conjunto de elementos relacionados. Luego, otras clases, más específicas, pueden heredar esta clase al agregar cada una de ellas lo que las hace únicas.

En el lenguaje de Java, a una clase que se hereda se le conoce como *superclase*. A la clase que recibe la herencia se le denomina *subclase*. Por lo tanto, una subclase es una versión especializada de una superclase. Hereda todas las variables y los métodos definidos por la superclase y agrega sus propios elementos únicos.

HABILIDAD FUNDAMENTAL 7.1

Fundamentos de la herencia

Java soporta la herencia al permitir que una clase incorpore a otra en su declaración. Esto se hace mediante el uso de la palabra clave **extends**. Así, la subclase se añade a (extiende) la superclase.

Empecemos con un ejemplo breve que ilustra varias de las funciones clave de la herencia. El siguiente programa crea una superclase llamada **FormaDosD**, que almacena el ancho y la altura de un objeto bidimensional, y una subclase llamada **Triang**. Observe cómo se usa la palabra clave **extends** para crear una subclase.

```
// Una jerarquía de clase simple.

// Una clase para objetos bidimensionales.
class FormaDosD {
  double ancho;
  double alto;

  void mostrarDim() {
    System.out.println("El ancho y alto son " +
                       ancho + " y " + alto);
  }
}

// Una subclase de FormaDosD para triángulos.
class Triang extends FormaDosD {
  String estilo;

  double área() {
    return ancho * alto / 2;
  }

  void mostrarEstilo() {
    System.out.println("El triángulo es " + estilo);
```

Triang hereda **FormaDosD**.

Triang puede hacer referencia a los miembros de **FormaDosD** como si fueran parte de **Triang**.

```
  }
}

class Formas {
  public static void main(String args[]) {
    Triang t1 = new Triang();
    Triang t2 = new Triang();

    t1.ancho = 4.0;
    t1.alto = 4.0;
    t1.estilo = "isósceles";

    t2.ancho = 8.0;
    t2.alto = 12.0;
    t2.estilo = "recto";

    System.out.println("Info para t1: ");
    t1.mostrarEstilo();
    t1.mostrarDim();
    System.out.println("El área es " + t1.area"());

    System.out.println();

    System.out.println("Info para t2: ");
    t2.mostrarEstilo();
    t2.mostrarDim();
    System.out.println("El área es " + t2.área());
  }
}
```

Todos los miembros de Triang están disponibles para los objetos de **Triang**, aun los heredados de **FormaDosD**.

A continuación se muestra la salida de este programa:

```
Info para t1:
El triángulo es isósceles
El ancho y alto son 4.0 y 4.0
El área es 8.0

Info para t2:
El triángulo es recto
El ancho y alto son 8.0 y 12.0
El área es 48.0
```

En este caso, **FormaDosD** define los atributos de una forma bidimensional "genérica" como un cuadrado, rectángulo, triángulo, etc. La clase **Triang** crea un tipo específico de **FormaDosD**; en este caso, un triángulo. La clase **Triang** incluye todo lo mismo que **FormaDosD** y agrega el campo

estilo, el método **área()** y el método **mostrarEstilo()**. En estilo se almacena una descripción del tipo de triángulo, **área()** calcula y regresa el área del triángulo y **mostrarEstilo()** despliega el estilo de triángulo.

Debido a que **Triang** incluye a todos los miembros de su superclase, **FormaDosD** puede tener acceso a **ancho** y **alto** dentro de **área()**. Además, dentro de **main()**, los objetos **t1** y **t2** pueden hacer referencia a **ancho** y **alto** directamente, como si fueran parte de **Triang**. En la figura 7.1 se describe conceptualmente la manera en que **FormaDosD** se incorpora en **Triang**.

Aunque **FormaDosD** es una superclase para **Triang**, es también una clase completamente independiente y autónoma. Ser la superclase de una subclase no significa que la superclase no pueda usarse de manera independiente. Por ejemplo, lo siguiente es perfectamente válido:

```
FormaDosD forma = new FormaDosD();

forma.ancho = 10;
forma.alto = 20;

forma.mostrarDim();
```

Por supuesto, un objeto de **FormaDosD** no tiene conocimiento de ninguna de sus subclases, ni tiene acceso a ellas.

A continuación se muestra la forma general de una declaración de clase que hereda una superclase:

```
class nombre-subclase extends nombre-superclase  {
  // cuerpo de la clase
}
```

Sólo es posible especificar una superclase para cualquier subclase que cree. Java no soporta la herencia de varias superclases en una sola subclase (lo cual es diferente a C++, en el que es posible heredar varias clases de base. Esté consciente de ello cuando convierta código de C++ a Java.) Sin embargo, sí puede crear una jerarquía de herencia en la que una subclase se convierta en la superclase de otra subclase. Por supuesto, ninguna clase puede ser una superclase de sí misma.

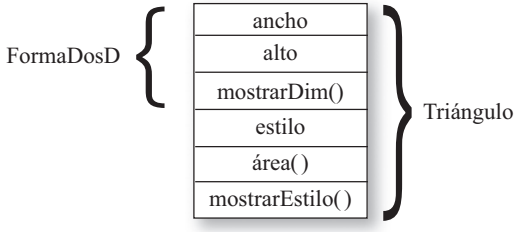


Figura 7.1 Descripción conceptual de la clase **Triang**.

Una ventaja importante de la herencia es que una vez que ha creado una superclase que define los atributos comunes a un conjunto de objetos, ésta puede usarse para crear cualquier número de subclases más específicas. Cada subclase puede adecuarse de manera precisa a su propia clasificación. Por ejemplo, he aquí otra subclase de **FormaDosD** que encapsula rectángulos:

```
// Una subclase de FormaDosD para rectángulos.
class Rectang extends FormaDosD {
  boolean esCuadrado() {
    if(ancho == alto) return true;
    return false;
  }

  double área() {
    return ancho * alto;
  }
}
```

La clase **Rectang** incluye **FormaDosD** y agrega el método **esCuadrado()**, que determina si el rectángulo es cuadrado, y **área()**, que calcula el área del rectángulo.

Acceso a miembros y herencia

Como aprendió en el módulo 6, a menudo la variable de instancia de una clase se declarará **private** para evitar su uso no autorizado o su modificación. La herencia de clase *no* sobrescribe la restricción de acceso de **private**. Por lo tanto, aunque una subclase incluye a todos los miembros de su superclase, ésta no puede acceder a los miembros de la superclase que se han declarado como **private**. Por ejemplo, si, como se muestra aquí, **ancho** y **alto** son private en **FormaDosD**, entonces **Triang** no podrá tener acceso a ellos.

```
// Los miembros private no se heredan.

// Este ejemplo no se compila.

// Una clase para objetos bidimensionales.
class FormaDosD {
  private double ancho;  // ahora son
  private double alto; // private

  void mostrarDim() {
    System.out.println("ancho y alto son " +
                       ancho + " y " + alto);
  }
}
```

```
// Una subclase de FormaDosD para Triangs.
class Triang extends FormaDosD {
  String estilo;

  double área() {
    return ancho * alto / 2; // ¡Error! no puede accederse
  }

  void mostrarEstilo() {
    System.out.println("El triángulo es " + estilo);
  }
}
```

No puede acceder a un miembro **private** de una superclase.

La clase **Triang** no se compilará porque la referencia a **ancho** y **alto** dentro del método **área()** origina una violación de acceso. Debido a que **ancho** y **alto** se declaran como **private**, sólo son accesibles para otros miembros de su propia clase. Las subclases no tienen acceso a ellos.

Recuerde que un miembro de clase que se ha declarado como **private** permanecerá privado para su clase, así que no es accesible para ningún código fuera de su clase, incluyendo las subclases.

Al principio, es posible que crea que el hecho de que una subclase no tenga acceso a los miembros private de su superclase, constituye una restricción seria que evita el uso de miembros private en muchas situaciones. Sin embargo, esto no es cierto. Como ya se explicó en el módulo 6, los programadores de Java suelen utilizar métodos de accesor para proporcionar acceso a los miembros privados de una clase. He aquí una versión rescrita de las clases **FormaDosD** y **Triang** que usan métodos para acceder a las variables de instancia privadas **ancho** y **alto**.

```
// Uso de métodos de accesor para establecer y obtener miembros private.

// Una clase para objetos bidimensionales.
class FormaDosD {
  private double ancho;  // ahora son
  private double alto;   // private

  // Métodos de accesor para ancho y alto.
  double obtenerAncho() { return ancho; }
  double obtenerAlto() { return alto; }
  void establecerAncho(double w) { ancho = w; }
  void establecerAlto(double h) { alto = h; }

  void mostrarDim() {
    System.out.println("El ancho y alto son " +
                       ancho + " y " + alto);
  }
}
```

Métodos de accesor para **ancho** y **alto**.

```
// Una subclase de FormaDosD para Triangs.
class Triang extends FormaDosD {
  String estilo;

  double área() {
    return obtenerAncho() * obtenerAlto() / 2;
  }

  void mostrarEstilo() {
    System.out.println("El triángulo es " + estilo);
  }
}

class Formas2 {
  public static void main(String args[]) {
    Triang t1 = new Triang();
    Triang t2 = new Triang();

    t1.establecerAncho(4.0);
    t1.establecerAlto(4.0);
    t1.estilo = "isósceles";

    t2.establecerAncho(8.0);
    t2.establecerAlto(12.0);
    t2.estilo = "recto";

    System.out.println("Info para t1: ");
    t1.mostrarEstilo();
    t1.mostrarDim();
    System.out.println("El área es " + t1.area());

    System.out.println();

    System.out.println("Info para t2: ");
    t2.mostrarEstilo();
    t2.mostrarDim();
    System.out.println("El área es " + t2.área());
  }
}
```

Use métodos de accesor proporcionados por la superclase.

Pregunte al experto

P: **¿Cuándo debo hacer que una variable de instancia sea private?**

R: No hay reglas inamovibles, pero he aquí dos principios generales: si utiliza una variable de instancia únicamente con métodos definidos dentro de su clase, entonces ésta debe hacerse private; si una variable de instancia debe estar dentro de ciertos límites, entonces debe ser private y quedar disponible sólo mediante métodos de accesor. De esta manera evitará que se le asignen valores que no son válidos.

Comprobación de avance

1. Cuando se crea una subclase, ¿cuál palabra clave se usa para incluir una superclase?
2. ¿Una subclase incluye a los miembros de su superclase?
3. ¿Una subclase tiene acceso a los miembros private de su superclase?

HABILIDAD FUNDAMENTAL 7.2

Constructores y herencia

En una jerarquía es posible que superclases y subclases tengan sus propios constructores. Esto plantea una pregunta importante: ¿cuál constructor es responsable de construir un objeto de la subclase? ¿El de la superclase, el de la subclase o ambos? La respuesta es la siguiente: el constructor de la superclase construye la parte de la superclase del objeto, y el de la subclase, la parte de esta última. Esto tiene sentido pues la superclase no conoce los elementos de una subclase o no tiene acceso a ellos. Por lo tanto, sus constructores deben estar separados. Los ejemplos anteriores dependen de los constructores predeterminados que Java crea automáticamente, de modo que no hay problema. Sin embargo, en la práctica, la mayor parte de las clases tienen constructores explícitos. A continuación se explica cómo manejar esta situación.

1. **extends**
2. Sí.
3. No.

Cuando sólo la subclase define un constructor, el proceso resulta muy sencillo: simplemente se construye el objeto de la subclase. La parte de la superclase del objeto se construye automáticamente usando su constructor predeterminado. Por ejemplo, he aquí una versión retrabajada de **Triang** que define un constructor. Asimismo, hace que **estilo** sea private pues ahora el constructor lo define.

```
// Agrega un constructor a Triang.

// Una clase para objetos bidimensionales.
class FormaDosD {
  private double ancho;  // ahora son
  private double alto;   // private

  // Métodos de accesor para ancho y alto.
  double obtenerAncho() { return ancho; }
  double obtenerAlto() { return alto; }
  void establecerAncho(double w) { ancho = w; }
  void establecerAlto(double h) { alto = h; }

  void mostrarDim() {
    System.out.println("El ancho y alto son " +
                       ancho + " y " + alto);
  }
}

// Una subclase de FormaDosD para Triangs.
class Triang extends FormaDosD {
  private String estilo;

  // Constructor
  Triang(String s, double w, double h) {
    establecerAncho(w); <-- Inicializa la parte Two DShape del objeto.
    establecerAlto(h);

    estilo = s;
  }

  double área() {
    return obtenerAncho() * obtenerAlto() / 2;
  }

  void mostrarEstilo() {
    System.out.println("El triángulo es " + estilo);
  }
}
```

```
Clase Forma tres{
  public static voind main (String args []) {
    Triangle t1 = new triangle ("isósceles", 4.0, 4.0);
    Triangle t2 = new triangle ("derecho", 8.0, 12.0);

    System.out.println (:Información para t1: ");
    t1.mostrarEstilo ();
    t1.mostrarDim ();
    System.out.println ("Área es" + t1.área());

    System.out.println();

    System.out.println (Información para t2: ");
    t2.mostrarEstilo ();
    t2.mostrarDim ();
    System.out.println ("Área es" + t2.área());
  }
}
```

En este caso, el constructor de **Triang** inicializa los miembros de **FormaDosD** que hereda junto con su propio campo **estilo**.

Cuando la subclase y la superclase definen constructores, el proceso se vuelve un poco más complicado porque ambos constructores deben ejecutarse. En este caso, debe usar otra de las palabras clave de Java, **súper**, que tiene dos formas generales. La primera llama a un constructor de la súperclase; la segunda se usa para que un miembro de una subclase acceda a un miembro de la superclase que se ha ocultado. A continuación analizaremos el uso de la primera.

Uso de súper para llamar a constructores de una súperclase

Una subclase puede llamar a un constructor definido por su superclase con el uso de la siguiente forma de **súper**:

súper(*lista-parámetros*);

Aquí, *lista-parámetros* especifica cualquier parámetro que el constructor de la superclase necesite. **súper( )** debe ser siempre la primera instrucción que se ejecute dentro de un constructor de subclase.

Para ver cómo se emplea **súper( )**, considere la versión de **FormaDosD** del siguiente programa. Ésta define un constructor que inicializa **ancho** y **alto**.

```
// Agrega constructores a FormaDosD.
class FormaDosD {
  private double ancho;
  private double alto;

  // Constructor con parámetros.
  FormaDosD(double w, double h) {
    ancho = w;
    alto = h;
  }

  // Métodos de accesor para ancho y alto.
  double obtenerAncho() { return ancho; }
  double obtenerAlto() { return alto; }
  void establecerAncho(double w) { ancho = w; }
  void establecerAlto(double h) { alto = h; }

  void mostrarDim() {
    System.out.println("El ancho y alto son " +
                       ancho + " y " + alto);
  }
}

// Una subclase de FormaDosD para Triangs.
class Triang extends FormaDosD {
  private String estilo;

  Triang(String s, double w, double h) {
    súper(w, h); // llama al constructor de la súperclase
    estilo = s;
  }

  double área() {
    return obtenerAncho() * obtenerAlto() / 2;
    }

    void mostrarEstilo() {
      System.out.println("El triángulo es " + estilo);
    }
  }
```

Use **super()** para ejecutar el constructor de **FormaDosD**.

```
class Formas4 {
  public static void main(String args[]) {
    Triang t1 = new Triang("isósceles", 4.0, 4.0);
  Triang t2 = new Triang("recto", 8.0, 12.0);

  System.out.println("Info para t1: ");
  t1.mostrarEstilo();
  t1.mostrarDim();
  System.out.println("El área es " + t1.area());

  System.out.println();

  System.out.println("Info para t2: ");
  t2.mostrarEstilo();
  t2.mostrarDim();
  System.out.println("El área es " + t2.area());
 }
}
```

Aquí, **Triang()** llama a **súper()** con los parámetros **w** y **b**. Esto hace que se llame al constructor de **FormaDosD()** que inicializa **ancho** y **alto** empleando estos valores. **Triang** ya no inicializa estos valores por sí mismo, sólo necesita inicializar el valor único de él: **estilo**. Esto deja a **FormaDosD** libre para construir su subobjeto de la manera que elija. Más aún, **FormaDosD** puede agregar una funcionalidad sobre la cual la subclase existente no tiene conocimiento, con lo que se evita que el código existente deje de funcionar.

Cualquier forma del constructor definida por la superclase puede ser llamada por **súper()**. El constructor ejecutado será el que coincida con los argumentos. Por ejemplo, aquí se presentan versiones expandidas de **FormaDosD** y **Triang** que incluyen constructores predeterminados y otros que toman un argumento.

```
// Agrega más constructores a FormaDosD.
class FormaDosD {
  private double ancho;
  private double alto;

  // Un constructor predeterminado.
  FormaDosD() {
    ancho = alto = 0.0;
  }
```

```
  // Constructor con parámetros.
  FormaDosD(double w, double h) {
    ancho = w;
    alto = h;
  }

  // Construye un objeto con ancho y alto iguales.
  FormaDosD(double x) {
    ancho = alto = x;
  }

  // Métodos de accesor para ancho y alto.
  double obtenerAncho() { return ancho; }
  double obtenerAlto() { return alto; }
  void establecerAncho(double w) { ancho = w; }
  void establecerAlto(double h) { alto = h; }

  void mostrarDim() {
    System.out.println("El ancho y alto son " +
                       ancho + " y " + alto);
  }
}

// Una subclase de FormaDosD para Triangs.
class Triang extends FormaDosD {
  private String estilo;

  // Un constructor predeterminado.
  Triang() {
    súper();
    estilo = "null";
  }

  // Constructor
  Triang(String s, double w, double h) {
    súper(w, h); // llama al constructor de la súperclase

    estilo = s;
  }

  // Construye un triángulo isósceles.
  Triang(double x) {
    súper(x); // llama al constructor de la súperclase
```

Use **super()** para llamar a las varias formas del constructor **FormaDosD**.

```
        estilo = "isósceles";

  }

    double área() {
      return obtenerAncho() * obtenerAlto() / 2;
    }

    void mostrarEstilo() {
      System.out.println("El triángulo es " + estilo);
    }
  }

  class Formas5 {
    public static void main(String args[]) {
      Triang t1 = new Triang();
      Triang t2 = new Triang("recto", 8.0, 12.0);
      Triang t3 = new Triang(4.0);

      t1 = t2;

      System.out.println("Info para t1: ");
      t1.mostrarEstilo();
      t1.mostrarDim();
      System.out.println("El área es " + t1.área());

      System.out.println();

      System.out.println("Info para t2: ");
      t2.mostrarEstilo();
      t2.mostrarDim();
      System.out.println("El área es " + t2.área());

      System.out.println();

      System.out.println("Info para t3: ");
      t3.mostrarEstilo();
      t3.mostrarDim();
      System.out.println("El área es " + t3.área());

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

Aquí se muestra la salida de este programa:

```
Info para t1:
El triángulo es recto
El ancho y alto son 8.0 y 12.0
El área es 48.0

Info para t2:
El triángulo es recto
El ancho y alto son 8.0 y 12.0
El área es 48.0

Info para t3:
El triángulo es isósceles
El ancho y alto son 4.0 y 4.0
El área es 8.0
```

Revisemos los conceptos clave detrás de **súper()**. Cuando una subclase llama a **súper()**, está llamando al constructor de su súperclase inmediata. Por lo tanto, **súper()** siempre hace referencia a la súperclase inmediatamente superior a la clase que llama. Esto es cierto aun en jerarquías de varios niveles. Además, **súper()** siempre debe ser la primera instrucción ejecutada dentro de un constructor de subclase.

Comprobación de avance

1. ¿De qué manera una subclase ejecuta un constructor de su superclase?

2. ¿Pueden pasarse parámetros mediante **súper()**?

3. ¿Puede estar **súper()** en cualquier lugar dentro de un constructor de una subclase?

1. Llama a s**úper()**.
2. Sí.
3. No, debe ser la primera instrucción ejecutada.

HABILIDAD FUNDAMENTAL 7.3

Uso de super para acceder a miembros de una superclase

Hay una segunda forma de **super** que actúa en cierto modo como **this**, excepto que siempre alude a la superclase de la subclase en la que es utilizada. Este uso tiene la siguiente forma general:

super.*miembro*

Aquí, *miembro* puede ser un método o una variable de instancia.

Esta forma de **super** es más aplicable a situaciones en las que los nombres de miembros de una subclase ocultan a los miembros con los mismos nombres en la superclase. Considere la siguiente jerarquía simple de clases:

```
// Uso de super para resolver el ocultamiento de nombres.
class A {
  int i;
}

// Crea una subclase al extender la clase A.
class B extends A {
  int i; // esta i oculta la i en A

  B(int a, int b) {
    super.i = a; // i in A
    i = b; // i in B
  }

  void mostrar() {
    System.out.println("i en la superclase: " + super.i);
    System.out.println("i en la subclase: " + i);
  }
}

class UseSuper {
  public static void main(String args[]) {
    B subOb = new B(1, 2);

    subOb.mostrar();
  }
}
```

Aquí, **super.i** se refiere a la **i** en **A**

Este programa despliega lo siguiente:

```
i en superclase: 1
i en subclase: 2
```

Aunque la variable de instancia **i** en **B** oculta la **i** en **A**, super permite el acceso a la **i** definida en la superclase. **super** también puede usarse para llamar métodos que están cultos por la subclase.

Proyecto 7.1 Extensión de la clase Automotor

CamionetaDemo.java

Para ilustrar la capacidad de la herencia, extenderemos la clase Automotor que se desarrolló por primera vez en el módulo 4. Como recordará, **Automotor** encapsula información acerca de automotores, incluyendo el número de pasajeros que puede transportar, la capacidad de su tanque de gasolina y el rango de consumo de gasolina. Podemos usar la clase **Automotor** como punto de partida para desarrollar clases más especializadas. Por ejemplo, un tipo de automotor es una camioneta. Un aspecto importante de una camioneta es su capacidad de carga, así que, para crear una clase **Camioneta**, puede extender **Automotor** agregando una variable de instancia que almacene la capacidad de carga. He aquí la versión de **Automotor** que lleva lo anterior a cabo. En este proceso, las variables de instancia en **Automotor** serán **private** y se proporcionarán métodos de accesor para obtener y establecer los valores.

Paso a paso

1. Cree un archivo llamado **CamionetaDemo.java** y copie la última implementación de **Automotor** del módulo 4 en el archivo.
2. Cree la clase **Camioneta** como se muestra aquí:

```
// Extiende Automotor para crear una especialización Camioneta.
class Camioneta extends Automotor {
  private int capcarga; // capacidad de carga en kilos

  // Este es un constructor para Camioneta.
  Camioneta(int p, int f, int m, int c) {
    /* Inicializa miembros de Automotor usando el
       constructor de Automotor. */
    super(p, f, m);

    capcarga = c;
  }

  // Métodos de accesor para capcarga.
  int obtenerCarga() { return capcarga; }
  void colocarCarga(int c) { capcarga = c; }
}
```

Aquí, **Camioneta** hereda **Automotor** agregando **capcarga**, **obtenerCarga()** y **colocarCarga()**. De manera que, **Camioneta** incluye todos los atributos generales de los automotores definidos por **Automotor**. Sólo necesita agregar los elementos que son únicos para su propia clase.

(continúa)

3. A continuación, haga que las variables de instancia de **Automotor** sean private, como se muestra aquí.

```
private int pasajeros;    // número de pasajeros
private int tanquegas;    // capacidad del tanque en litro
private int kpl;          // consumo de gasolina en km por litro
```

4. He aquí todo el programa que demuestra la clase **Camioneta**.

```
// Construye una subclase de Automotor para camionetas.
class Automotor {
  private int pasajeros;    // número de pasajeros
  private int tanquegas;    // capacidad del tanque en litro
  private int kpl;          // consumo de gasolina en km por litro

  // Este es un constructor para Automotor.
  Automotor(int p, int f, int m) {
    pasajeros = p;
    tanquegas = f;
    kpl = m;
  }

  // Regresa el rango.
  int rango() {
    return kpl * tanquegas;
  }

  // calcula la gasolina necesaria para una distancia.
  double gasnecesaria(int km) {
    return (double) km / kpl;
  }

  // Métodos de acceso para variables de instancia.
  int obtenerPasajeros() { return pasajeros; }
  void establecerPasajeros(int p) { pasajeros = p; }
  int obtenerTanquegas() { return tanquegas; }
  void establecerTanquegas(int f) { tanquegas = f; }
  int obtenerKpl() { return kpl; }
  void establecerKpl(int m) { kpl = m; }

}

// Extiende Automotor para crear una especialización Camioneta.
class Camioneta extends Automotor {
  private int capcarga; // capacidad de carga en kilos

  // Este es un constructor para Camioneta.
  Camioneta(int p, int f, int m, int c) {
```

```
    /* Inicializa miembros de Automotor usando el
       constructor de Automotor. */
    super(p, f, m);

    capcarga = c;
  }

  // Métodos de accesor para capcarga.
  int obtenerCarga() { return capcarga; }
  void colocarCarga(int c) { capcarga = c; }
}

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

    // construye algunas camionetas
    Camioneta semi = new Camioneta(2, 800, 2, 20000);
    Camioneta pickup = new Camioneta(3, 100, 4, 1000);
    double litros;
    int dist = 252;

    litros = semi.gasnecesaria(dist);

    System.out.println("Un semiremolque puede cargar " + semi.
obtenerCarga() + " kilos.");
    System.out.println("Para recorrer " + dist + " kms necesita " +
                       litros + " litros de gasolina.\n");

    litros = pickup.gasnecesaria(dist);

    System.out.println("Una pickup puede cargar " + pickup.obtenerCarga() +
                       " kilos.");
    System.out.println("Para recorrer " + dist + " kms la pickup necesita " +
                       litros + " litros de gasolina.");
  }
}
```

5. Aquí se muestra la salida de este programa:

```
Un semiremolque puede cargar 20000 kilos.
Para recorrer 252 kms necesita 126.0 litros de gasolina.

Una pickup puede cargar 1000 kilos.
Para recorrer 252 kms la pickup necesita 63.0 litros de gasolina.
```

(continúa)

6. Es posible derivar muchos otros tipos de clases de **Automotor**. Por ejemplo, el siguiente esqueleto crea una clase TodoTerreno que almacena el terreno despejado por el vehículo.

```
// Crea una clase TodoTerreno de Automotor
class TodoTerreno extends Automotor {
  private int terrenoDespejado; // terreno despejado en cm

  // ...
}
```

Lo importante es que una vez que haya creado una superclase que defina los aspectos generales de un objeto, esa superclase puede heredarse para formar clases especializadas. Cada subclase simplemente agrega sus atributos propios y únicos, lo cual constituye el aspecto esencial de la herencia.

HABILIDAD FUNDAMENTAL 7.4

Creación de una jerarquía de varios niveles

Hasta este momento hemos usado jerarquías simples de clases que sólo constan de una súperclase y una subclase. Sin embargo, puede construir jerarquías que contengan todas las capas de herencia que desee. Como ya se mencionó, es perfectamente aceptable usar una subclase como súperclase de otra. Por ejemplo, dadas tres clases llamadas **A**, **B** y **C**, **C** puede ser una subclase de **B**, que es una subclase de **A**. Cuando este tipo de situación ocurre, cada subclase hereda todos los rasgos encontrados en sus súperclases. En este caso, **C** hereda todos los aspectos de **B** y **A**.

Para ver la utilidad de una jerarquía de varios niveles, analice el siguiente programa: en él, la subclase **Triang** se usa como superclase para crear la subclase llamada **ColorTriang**. Ésta hereda todos los rasgos de **Triang** y de **FormaDosD** y agrega un campo llamado **color**, el cual incluye el color del triángulo.

```
// Una jerarquía de varios niveles.
class FormaDosD {
  private double ancho;
  private double alto;

  // Un constructor predeterminado.
  FormaDosD() {
    ancho = alto = 0.0;
  }

  // Un constructor con parámetros.
  FormaDosD(double w, double h) {
    ancho = w;
    alto = h;
  }
```

```
  // Construye un objeto de ancho y alto iguales.
  FormaDosD(double x) {
    ancho = alto = x;
  }

  // Métodos de accesor para ancho y alto.
  double obtenerAncho() { return ancho; }
  double obtenerAlto() { return alto; }
  void establecerAncho(double w) { ancho = w; }
  void establecerAlto(double h) { alto = h; }

  void mostrarDim() {
    System.out.println("El ancho y alto son " +
                       ancho + " y " + alto);
  }
}

// Extiende FormaDosD.
class Triang extends FormaDosD {
  private String estilo;

  // Un constructor predeterminado.
  Triang() {
    súper();
    estilo = "null";
  }

  Triang(String s, double w, double h) {
    súper(w, h); // llama al constructor de la súperclase

    estilo = s;
  }

  // Construye un triángulo isósceles.
  Triang(double x) {
    súper(x); // llama al constructor de la súperclase

    estilo = "isósceles";
  }

  double área() {
    return obtenerAncho() * obtenerAlto() / 2;
  }
```

```
  void mostrarEstilo() {
    System.out.println("El triángulo es " + estilo);
  }
}

// Extiende Triang.
class ColorTriang extends Triang {
  private String color;

  ColorTriang(String c, String s,
              double w, double h) {
    super(s, w, h);

    color = c;
  }

  String getColor() { return color; }

  void mostrarColor() {
    System.out.println("El color es " + color);
  }
}

class Formas6 {
  public static void main(String args[]) {
    ColorTriang t1 =
         new ColorTriang("Azul", "recto", 8.0, 12.0);
    ColorTriang t2 =
         new ColorTriang("Rojo", "isósceles", 2.0, 2.0);

    System.out.println("Info para t1: ");
    t1.mostrarEstilo();
    t1.mostrarDim();
    t1.mostrarColor();
    System.out.println("El área es " + t1.área());

    System.out.println();

    System.out.println("Info para t2: ");
    t2.mostrarEstilo();
    t2.mostrarDim();
    t2.mostrarColor();
    System.out.println("El área es " + t2.área());
  }
}
```

ColorTriang hereda **Triang**, que es un descendiente de **FormaDosD**, de modo que **ColorTriang** incluye todos los miembros de **Triang** y de **FormaDosD**.

Un objeto de **ColorTriang** puede llamar a todos los métodos definidos por sí mismo y su superclase.

Esta es la salida de este programa:

```
Info para t1:
El triángulo es recto
El ancho y alto son 8.0 y 12.0
El color es Azul
El área es 48.0

Info para t2:
El triángulo es isósceles
El ancho y alto son 2.0 y 2.0
El color es Rojo
El área es 2.0
```

Debido a la herencia, **ColorTriang** puede usar las clases previamente definidas de **Triang** y **FormaDosD** agregando sólo la información adicional que necesita para su aplicación propia y específica. Esto es parte del valor de la herencia: permite el reciclaje del código.

En este ejemplo se ilustra otro tema importante: **super()** siempre alude al constructor en la súperclase más cercana. El **súper()** de **ColorTriang** llama al constructor de **Triang**. El **súper()** de **Triang** llama al constructor de **FormaDosD**. En una jerarquía de clase, si el constructor de una súperclase requiere parámetros, entonces todas las subclases deben pasar estos parámetros "hacia arriba". Esto es cierto sin importar si la subclase necesita parámetros propios o no.

HABILIDAD FUNDAMENTAL 7.5

¿Cuándo se llama a los constructores?

En el análisis anterior sobre la herencia y las jerarquías de clases, es probable que se haya preguntado lo siguiente: cuando un objeto de una subclase se crea, ¿qué constructor se ejecuta primero?, ¿el de la subclase o el definido por la súperclase? Por ejemplo, dada una subclase llamada **B** y una súperclase llamada **A**, ¿se llama al constructor de **A** antes que al de **B**, o viceversa? La respuesta es que en una jerarquía de clase, los constructores son llamados en orden de derivación, de la súperclase a la subclase. Más aún, como **súper()** debe ser la primera instrucción que se ejecute en un constructor de subclase, este orden es el mismo sin importar si se usa **súper()** o no. Si no se usa **súper()**, entonces se ejecutará el constructor predeterminado (sin parámetros) de cada superclase. El siguiente programa ilustra el momento en que se ejecutan los constructores:

```
// Demuestra el momento en que se llama a los constructores.

// Crea una superclase.
class A {
  A() {
    System.out.println("Construyendo A.");
  }
}
```

```
// Crea una subclase al extender la clase A.
class B extends A {
  B() {
    System.out.println("Construyendo B.");
  }
}

// Crea otra subclase extendiendo B.
class C extends B {
  C() {
    System.out.println("Construyendo C.");
  }
}

class OrdenDeConstruc {
  public static void main(String args[]) {
    C c = new C();
  }
}
```

Ésta es la salida de este programa:

```
Construyendo A.
Construyendo B.
Construyendo C.
```

Como puede ver, los constructores son llamados en orden de derivación.

Si lo considera, resulta lógico que los constructores se ejecuten en este orden. Debido a que la súperclase no conoce ninguna subclase, cualquier inicialización que necesite realizar se encuentra separada, y tal vez sea el prerequisito, de cualquier inicialización realizada por la subclase. Por lo tanto, debe ejecutarse primero.

HABILIDAD FUNDAMENTAL 7.6

Referencias a súperclases y objetos de subclases

Como ya lo sabe, Java es un lenguaje orientado fuertemente a tipos. Aparte de las conversiones estándar y las promociones automáticas que se aplican a sus tipos primitivos, la compatibilidad de tipos está estrictamente impuesta. Así que, por lo general, una variable de referencia para un tipo de clase no tiene la capacidad de hacer referencia a un objeto de otro tipo de clase. Por ejemplo, considere el siguiente programa:

```
// Esto no se compilará.
class X {
  int a;
```

```
  X(int i) { a = i; }
}

class Y {
  int a;

  Y(int i) { a = i; }
}

class RefIncompatible {
  public static void main(String args[]) {
    X x = new X(10);
    X x2;
    Y y = new Y(5);

    x2 = x; // OK, ambos son del mismo tipo

    x2 = y; // Error, no son del mismo tipo
  }
}
```

Aquí, aunque las clases **X** y **Y** son físicamente iguales, no es posible asignar a un objeto **Y** una referencia a **X** porque tienen tipos diferentes. En general, una variable de referencia a objeto sólo puede hacer referencia a los objetos de su tipo.

Sin embargo, existe una excepción importante en cuanto a la estricta imposición de tipo de Java. A una variable de referencia de una súperclase puede asignársele una referencia a cualquier subclase derivada de esa súperclase. He aquí un ejemplo:

```
// Una referencia a súperclase puede hacer referencia a un objeto de una
subclase.
class X {
  int a;

  X(int i) { a = i; }
}

class Y extends X {
  int b;

  Y(int i, int j) {
    súper(j);
    b = i;
  }
}
```

```
class RefSupSub {
  public static void main(String args[]) {
    X x = new X(10);
    X x2;
    Y y = new Y(5, 6);

    x2 = x; // OK, ambas son del mismo tipo
    System.out.println("x2.a: " + x2.a);

    x2 = y; // Todavía Ok porque Y se deriva de X
    System.out.println("x2.a: " + x2.a);

    // las referencias a X sólo saben acerca de los miembros de X
    x2.a = 19; // OK
//    x2.b = 27; // Error, no tiene un miembro b
  }
}
```

OK porque **Y** es una subclase de **X**; por lo tanto, **x2** puede hacer referencia a **y**.

Aquí, **Y** se deriva ahora de **X** por lo que se permite asignar a **x2** una referencia a un objeto de **Y**.

Es importante comprender que el tipo de variable de referencia (no el tipo del objeto al que se hace referencia) es el que determina el miembro al que puede accederse. Es decir, cuando una referencia a un objeto de una subclase se asigna a una variable de referencia a una súperclase, usted sólo tendrá acceso a las partes del objeto definidas por la súperclase. Por tal motivo, **x2** no puede acceder a **b** aunque haga referencia a un objeto de **Y**. Si lo considera, esto último resulta lógico, pues la superclase no sabe a cuál subclase añadirlo. Por ello, la última línea del código está comentada para eliminarla.

Aunque el análisis anterior parece un poco esotérico, tiene algunas aplicaciones prácticas importantes. Una de ellas se describe a continuación. La otra se analizará más adelante en este módulo, cuando se estudie la sobrescritura de métodos.

Un lugar importante en el que las referencias a subclases son asignadas a variables de superclases es cuando los constructores en una jerarquía de clase son llamados. Es común que una clase defina a un constructor que tome un objeto de la clase como parámetro. Ello permite que la clase construya una copia de un objeto. Las subclases de esta clase pueden aprovechar esta función. Por ejemplo, considere las siguientes versiones de **FormaDosD** y **Triang**. Ambas agregan constructores que toman a un objeto como parámetro.

```
class FormaDosD {
  private double ancho;
  private double alto;
```

```
  // Un constructor predeterminado.
  FormaDosD() {
    ancho = alto = 0.0;
  }

  // Un constructor con parámetros.
  FormaDosD(double w, double h) {
    ancho = w;
    alto = h;
  }

  // Construye objeto con ancho y alto iguales.
  FormaDosD(double x) {
    ancho = alto = x;
  }

  // Construye un objeto a partir de un objeto.
  FormaDosD(FormaDosD ob) {
    ancho = ob.ancho;
    alto = ob.alto;
  }

  // Métodos de accesor para ancho y alto.
  double obtenerAncho() { return ancho; }
  double obtenerAlto() { return alto; }
  void establecerAncho(double w) { ancho = w; }
  void establecerAlto(double h) { alto = h; }

  void mostrarDim() {
    System.out.println("El ancho y alto son " +
                       ancho + " and " + alto);
  }
}

// Una subclase de FormaDosD para Triangs.
class Triang extends FormaDosD {
  private String estilo;

  // Un constructor predeterminado.
  Triang() {
    súper();
    estilo = "null";
  }
```

Construye un objeto a partir de un objeto.

```
  // Constructor para Triang.
  Triang(String s, double w, double h) {
    súper(w, h); // llama a constructor de superclase

    estilo = s;
  }

  // Construye un triángulo isósceles.
  Triang(double x) {
    súper(x); // llama a constructor de superclase

    estilo = "isósceles";
  }

  // Construye un objeto a partir de un objeto.
  Triang(Triang ob) {
    súper(ob); // pasa un objeto al constructor de FormaDosD
    estilo = ob.estilo;
  }

  double área() {
    return obtenerAncho() * obtenerAlto() / 2;
  }

  void mostrarEstilo() {
    System.out.println("El triángulo es " + estilo);
  }
}

class Formas7 {
  public static void main(String args[]) {
    Triang t1 =
         new Triang("recto", 8.0, 12.0);

    // hace una copia de t1
    Triang t2 = new Triang(t1);

    System.out.println("Info para t1: ");
    t1.mostrarEstilo();
    t1.mostrarDim();
    System.out.println("El área es " + t1.área());

    System.out.println();
```

Pasa una referencia a **Triang** al constructor de **FormaDosD**.

```
    System.out.println("Info para t2: ");
    t2.mostrarEstilo();
    t2.mostrarDim();
    System.out.println("El área es " + t2.área());
  }
}
```

En este programa, **t2** se construye a partir de **t1** y es, por lo tanto, idéntico. Aquí se muestra la salida:

```
Info para t1:
El triángulo es recto
El ancho y alto son 8.0 and 12.0
El área es 48.0

Info para t2:
El triángulo es recto
El ancho y alto son 8.0 and 12.0
El área es 48.0
```

Ponga especial atención a este constructor de **Triang**:

```
// Construye un objeto a partir de un objeto.
Triang(Triang ob) {
  super(ob); // pasa un objeto al constructor de FormaDosD
  estilo = ob.estilo;
}
```

Recibe un objeto de tipo **Triang** y pasa ese objeto (a través de **súper**) a este constructor de **FormaDosD**:

```
// Construye un objeto a partir de un objeto.
FormaDosD(FormaDosD ob) {
  ancho = ob.ancho;
  alto = ob.alto;
}
```

La clave es que **FormaDosD()** está esperando un objeto de **FormaDosD**; sin embargo, **Triang()** lo pasa a un objeto de **Triang**. La razón de que esto funcione es que, como se explicó, una referencia a una superclase puede hacer referencia a un objeto de una subclase. Por consiguiente, es perfectamente aceptable pasar a **FormaDosD()** una referencia a un objeto de una clase derivada de **FormaDosD**. Debido a que el constructor de **FormaDosD** sólo inicializa las partes del objeto de la subclase que son miembros de **FormaDosD**, no importa que el objeto contenga también otros miembros agregados por las clases derivadas.

Comprobación de avance

1. ¿Puede usarse una subclase como superclase de otra subclase?
2. En una jerarquía de clases, ¿en qué orden se llama a los constructores?
3. Dado que **Jet** extiende a **Aeroplano**, ¿puede **Aeroplano** hacer referencia a un objeto de **Jet**?

HABILIDAD FUNDAMENTAL 7.7

Sobrescritura de métodos

En una jerarquía de clases, cuando un método de una subclase tiene el mismo tipo de regreso y la misma firma que un método de una superclase, se dice que el método de la subclase *sobrescribe* al de la superclase. Cuando un método sobrescrito es llamado desde el interior de una subclase, dicho método siempre hará referencia a la versión del método definida por la subclase. La versión del método definida por la superclase estará oculta. Considere lo siguiente:

```
// Sobrescritura de métodos.
class A {
  int i, j;
  A(int a, int b) {
    i = a;
    j = b;
  }

  // despliega i y j
  void mostrar() {
    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;
  }
```

1. Sí.
2. Los constructores se llaman en orden de derivación.
3. Sí. En todos los casos, la referencia a una superclase puede hacer referencia a un objeto de una subclase, pero no a la inversa.

```
  // despliega k- esto sobrescribe mostrar() en A
  void mostrar() {
    System.out.println("k: " + k);
  }
}

class Sobresc {
  public static void main(String args[]) {
    B subOb = new B(1, 2, 3);

    subOb.mostrar(); // esto llama a mostrar() en B
  }
}
```

Este **mostrar()** en **B** sobrescribe el definido en **A**.

La salida producida por este programa se muestra aquí:

```
k: 3
```

Cuando se invoca a **mostrar()** en un objeto de tipo **B**, se usa la versión de **mostrar()** definida dentro de **B**. Es decir, la versión de **mostrar()** dentro de **B** sobrescribe a la versión declarada en **A**.

Si quiere acceder a la versión de la superclase de un método sobrescrito, puede hacerlo mediante el uso de **super**. Por ejemplo, en esta versión de **B**, la versión de la superclase de **mostrar()** se invoca dentro de la versión de la subclase. Esto permite que todas las variables de instancia se desplieguen.

```
class B extends A {
  int k;

  B(int a, int b, int c) {
    súper(a, b);
    k = c;
  }
  void mostrar() {
    súper.mostrar(); // esto llama al mostrar() de A
    System.out.println("k: " + k);
  }
}
```

Se usa súper para llamar a la versión de **mostrar()** definida por la superclase **A**.

Si sustituye esta versión de **mostrar()** en el programa anterior, verá entonces la siguiente salida:

```
i y j; 1 2
k: 3
```

Aquí, **super.mostrar()** llama a la versión de la superclase de **mostrar()**.

La sobrescritura de métodos *sólo* ocurre cuando los tipos de regreso y las firmas de los dos métodos son idénticos. Si no lo son, entonces los dos métodos simplemente están sobrecargados. Por ejemplo, considere esta versión modificada del ejemplo anterior:

```
/* Los métodos con diferentes firmas de tipo están
   sobrecargados y no se sobrescriben. */
class A {
  int i, j;

  A(int a, int b) {
    i = a;
    j = b;
  }

  // despliega i y j
  void mostrar() {
    System.out.println("i y j: " + i + " " + j);
  }
}

// Crea una subclase al extender la clase A.
class B extends A {
  int k;

  B(int a, int b, int c) {
    super(a, b);
    k = c;
  }

  // sobrecarga mostrar()
  void mostrar(String msj) {
    System.out.println(msj + k);
  }
}

class Sobrecarga {
  public static void main(String args[]) {
    B subOb = new B(1, 2, 3);

    subOb.mostrar("Esto es k: "); // esto llama a mostrar() en B
    subOb.mostrar(); // esto llama a mostrar() en A
  }
}
```

Debido a que las firmas son diferentes, este **mostrar( )** simplemente sobrecarga **mostrar( )** en la superclase **A**.

La salida producida por este programa se muestra a continuación:

```
Esto es k: 3
i y j: 1 2
```

La versión de **mostrar( )** en **B** toma un parámetro de cadena. Esto hace que su firma sea diferente a la de **A**, la cual no toma parámetros. Por lo tanto, no se presenta sobrescritura (ni ocultamiento de nombre).

HABILIDAD FUNDAMENTAL 7.8

Los métodos sobrescritos soportan polimorfismo

Aunque los ejemplos de la sección anterior demuestran la mecánica de la sobrescritura de métodos, no enseñan todas sus ventajas. Por supuesto, si no hubiera en la sobrescritura de métodos nada más que una convención de espacio de nombre, entonces se trataría, cuando mucho, de una curiosidad interesante, pero con escaso valor real. Sin embargo, éste no es el caso. La sobrescritura de métodos integra la base de uno de los conceptos más importantes de Java: *el despacho dinámico de métodos*. Se trata del mecanismo mediante el cual una llamada a un método sobrescrito se resuelve en el tiempo de ejecución y no en el de compilación. El despacho dinámico de métodos resulta importante porque es cómo Java implementa el polimorfismo en tiempo de ejecución.

Empecemos por repetir un principio importante: una variable de referencia a una superclase puede hacer referencia a un objeto de una subclase. Java emplea esto para resolver llamadas a métodos sobrescritos en tiempo de ejecución. He aquí cómo: cuando se llama a un método sobrescrito mediante una referencia a una superclase, Java determina qué versión del método ejecutar con base en el tipo de objeto al que se está haciendo referencia en el momento en que la llamada ocurre. Por lo tanto, la determinación se hace en el tiempo de ejecución. Cuando se hace referencia a tipos diferentes de objetos, se llama a distintas versiones de un método sobrescrito. En otras palabras, *es el tipo de objeto al que se hace referencia* (no el tipo de variable de referencia) lo que determina qué versión de un método sobrescrito se ejecuta. En consecuencia, si una superclase contiene un método sobrescrito por una subclase, entonces cuando se hace referencia a diferentes tipos de objeto mediante una variable de referencia a superclase, diferentes versiones del método se ejecutan.

He aquí un ejemplo que ilustra el despacho dinámico de métodos.

```
// Demuestra despacho dinámico de método.

class Sup {
  void quien() {
    System.out.println("quien() en Sup");
  }
}
```

```
class Sub1 extends Sup {
  void quien() {
    System.out.println("quien() en Sub1");
  }
}

class Sub2 extends Sup {
  void quien() {
    System.out.println("quien() en Sub2");
  }
}

class DispDinDemo {
  public static void main(String args[]) {
    Sup superOb = new Sup();
    Sub1 subOb1 = new Sub1();
    Sub2 subOb2 = new Sub2();

    Sup supRef;

    supRef = superOb;
    supRef.quien();

    supRef = subOb1;
    supRef.quien();

    supRef = subOb2;
    supRef.quien();
  }
}
```

En cada caso, la versión de **quien()** a la cual se llama se determina en tiempo de ejecución mediante el tipo de objeto al que se hace referencia.

La salida del programa se muestra aquí:

```
quien() en Sup
quien() en Sub1
quien() en Sub2
```

Este programa crea una superclase llamada **Sup** y dos subclases de ella, llamadas **Sub1** y **Sub2**. **Sup** declara un método llamado **quien()**, y la subclase lo sobrescribe. Dentro del método **main()**, se declaran objetos de tipo **Sup**, **Sub1** y **Sub2**. Además, se declara una referencia de tipo **Sup** llamada **supRef**. Luego el programa asigna una referencia a **supRef** a cada tipo de objeto y usa esta referencia para llamar a **quien()**. Como la salida lo muestra, la versión de **quien()** que se ejecuta está determinada por el tipo de objeto al que se está haciendo referencia en el tiempo de la llamada, y no por el tipo de clase de **supRef**.

Pregunte al experto

P: **Los métodos sobrescritos de Java tienen un aspecto similar a las funciones virtuales de C++. ¿Existe alguna similitud?**

R: Sí. Los lectores familiarizados con C++ reconocerán que los métodos sobrescritos de Java tienen un propósito equivalente y operan de manera similar a las funciones virtuales de C++.

¿Por qué los métodos se sobrescriben?

Como ya se afirmó, los métodos sobrescritos permiten a Java soportar el polimorfismo en tiempo de ejecución. El polimorfismo es esencial en la programación orientada a objeto por una razón: permite que una clase general especifique los métodos que serán comunes a todos sus derivados, mientras permite que las subclases definan la implementación específica de alguno o de todos esos métodos. Los métodos sobrescritos constituyen otra manera en la que Java implementa el aspecto "una interfaz, varios métodos" del polimorfismo.

Parte de la clave para la aplicación exitosa del polimorfismo es la comprensión de que las superclases y las subclases forman una jerarquía que va de una menor a una mayor especialización. Utilizada correctamente, la superclase proporciona todos los elementos que una subclase puede usar directamente. Asimismo, define los métodos que la clase derivada puede implementar por su cuenta, lo cual permite a la subclase la flexibilidad de definir sus propios métodos; sin embargo, impone todavía una interfaz consistente. De ahí que, al combinar la herencia con los métodos sobrescritos, una superclase puede definir la forma general de los métodos que utilizarán todas sus subclases.

Aplicación de la sobrescritura de métodos a FormaDosD

Para comprender mejor los alcances de la sobrescritura de métodos, la aplicaremos a la clase **FormaDosD**. En los ejemplos anteriores, cada clase derivada de **FormaDosD** definía un método llamado **área()**, lo cual sugiere que sería mejor hacer que **area()** fuera parte de la clase **FormaDosD**, con lo que se permitiría que cada subclase la sobrescribiera. Esto define la manera en que se calcula el área para el tipo de forma que encapsula la clase. El siguiente programa lleva a cabo todo lo anterior. Por conveniencia, también agrega un campo de nombre a **FormaDosD**. (Esto facilita la escritura de programas de demostración.)

```
// Uso del despacho dinámico de métodos.
class FormaDosD {
  private double ancho;
  private double alto;
  private String nombre;
```

```
  // Un constructor predeterminado.
  FormaDosD() {
    ancho = alto = 0.0;
    nombre = "null";
  }

  // Constructor con parámetros.
  FormaDosD(double w, double h, String n) {
    ancho = w;
    alto = h;
    nombre = n;
  }

  // Objeto de contructor con ancho y alto iguales.
  FormaDosD(double x, String n) {
    ancho = alto = x;
    nombre = n;
  }

  // Construye un objeto a partir de un objeto.
  FormaDosD(FormaDosD ob) {
    ancho = ob.ancho;
    alto = ob.alto;
    nombre = ob.nombre;
  }

  // Métodos de accesor para ancho y alto.
  double obtenerAncho() { return ancho; }
  double obtenerAlto() { return alto; }
  void establecerAncho(double w) { ancho = w; }
  void establecerAlto(double h) { alto = h; }

  String obtenerNombre() { return nombre; }

  void mostrarDim() {
    System.out.println("El ancho y alto son " +
                       ancho + " y " + alto);
  }

  double área() {
    System.out.println("área() debe sobrescribirse");
    return 0.0;
  }
}
```

El método **área()** definido por **FormaDosD**.

```
// Una subclase de FormaDosD para Triangs.
class Triang extends FormaDosD {
  private String estilo;

  // Un constructor predeterminado.
  Triang() {
    súper();
    estilo = "null";
  }

  // Constructor para Triang.
  Triang(String s, double w, double h) {
    súper(w, h, "un triángulo");

    estilo = s;
  }

  // Construye un triángulo isósceles.
  Triang(double x) {
    súper(x, "un triángulo"); // llama a constructor de súperclase

    estilo = "isósceles";
  }

  // Construye un objeto a partir de un objeto.
  Triang(Triang ob) {
    súper(ob); // pasa un objeto al constructor de FormaDosD
    estilo = ob.estilo;
  }

  // Sobrescribe área() para Triang.
  double área() {
    return obtenerAncho() * obtenerAlto() / 2;
  }

  void mostrarEstilo() {
    System.out.println("El triángulo es " + estilo);
  }
}

// Una subclase de FormaDosD para rectángulos.
class Rectang extends FormaDosD {
  // Un constructor predeterminado.
  Rectang() {
    súper();
  }
```

Sobrescribe **área()** con **Triang**.

```
  // Constructor para un rectángulo.
  Rectang(double w, double h) {
    súper(w, h, "un rectángulo"); // llama a un constructor de súperclase
  }

  // Construye un cuadrado.
  Rectang(double x) {
    súper(x, "un rectángulo"); // llama a un constructor de superclase
  }

  // Construye un objeto a partir de un objeto.
  Rectang(Rectang ob) {
    súper(ob); // pasa el objeto al constructor de FormaDosD
  }

  boolean esCuadrado() {
    if(obtenerAncho() == obtenerAlto()) return true;
    return false;
  }

  // Sobrescribe área() con Rectang.
  double área() {
    return obtenerAncho() * obtenerAlto();
  }
}

class FormasDin {
  public static void main(String args[]) {
    FormaDosD formas[] = new FormaDosD[5];

    formas[0] = new Triang("recto", 8.0, 12.0);
    formas[1] = new Rectang(10);
    formas[2] = new Rectang(10, 4);
    formas[3] = new Triang(7.0);
    formas[4] = new FormaDosD(10, 20, "genérico");

    for(int i=0; i < formas.length; i++) {
      System.out.println("El objeto es " + formas[i].obtenerNombre());
      System.out.println("El área es " + formas[i].área());
```

Sobrescribe **área** con **Rectang**.

Se llama la versión apropiada de **área()** para cada forma.

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

Ésta es la salida de este programa:

```
El objeto es un triángulo
El área es 48.0

El objeto es un rectángulo
El área es 100.0

El objeto es un rectángulo
El área es 40.0

El objeto es un triángulo
El área es 24.5

El objeto es gen/rico
área() debe sobrescribirse
El área es 0.0
```

Examinemos de cerca este programa. En primer lugar, como se explicó, **área( )** es ahora parte de la clase **FormaDosD** y es sobrescrita por **Triang** y **Rectang**. Dentro de **FormaDosD**, se ha dado a **área( )** una implementación de marcador de posición que simplemente informa al usuario que una subclase debe sobrescribir este método. Cada sobrescritura de **área( )** proporciona una implementación que resulta adecuada para el tipo de objeto encapsulado por la subclase. Por lo tanto, si fuera a implementar una clase Elipse, por ejemplo, entonces **área( )** tendría que calcular el **área** de una elipse.

Hay otro elemento importante en el programa anterior: observe en **main( )** que formas está declarada como una matriz de objetos de **FormaDosD**. Sin embargo, los elementos de esta matriz tienen asignadas referencias a **Triang**, **Rectang** y **FormaDosD**. Esto es válido porque, como se explicó, una referencia a superclase puede hacer referencia a un objeto de una subclase. Luego, el programa recorre en ciclo la matriz desplegando información acerca de cada objeto. Aunque es muy simple, lo anterior ilustra la capacidad de la herencia y la sobrescritura de métodos. El tipo de objeto al que hace referencia una variable de referencia está determinado en tiempo de ejecución y se actúa sobre él de manera acorde. Si un objeto se deriva de **FormaDosD**, entonces se puede obtener su área al llamar a **área( )**. La interfaz para esta operación es la misma sin importar el tipo de forma que se esté usando.

Comprobación de avance

1. ¿Qué es la sobrescritura de métodos?
2. ¿Por qué es importante la sobrescritura de métodos?
3. Cuando se llama a un método sobrescrito mediante una referencia a una superclase, ¿cuál versión del método se ejecuta?

HABILIDAD FUNDAMENTAL 7.9

Uso de clases abstractas

En ocasiones es posible que desee crear una superclase que defina sólo una forma generalizada que será compartida por todas sus subclases, dejando a cada subclase el llenado de los detalles. Este tipo de clase determina la naturaleza del método que la subclase debe implementar pero que, por sí solo, no proporciona una implementación de uno o más de estos métodos. Una manera en la que esta situación ocurre es cuando una superclase no puede crear una implementación significativa para un método. Éste es el caso de la versión de **FormaDosD** usada en el ejemplo anterior. La definición de **área()** es simplemente un marcador de posición, es decir, no calculará ni desplegará el área de ningún tipo de objeto.

Cuando cree sus propias bibliotecas de clases notará que no es raro que un método no tenga una definición significativa en el contexto de su superclase. Puede manejar esta situación de dos maneras: la primera, como se mostró en el ejemplo anterior, consiste simplemente en hacer que reporte un mensaje cálido. Aunque este método puede resultar útil en ciertas situaciones (como la depuración), no suele ser apropiado. Puede haber métodos que requieran ser sobrescritos por la subclase para que ésta tenga algún significado. Considere la clase **Triang**: no tiene significado si **área()** no está definida. En este caso, deseará contar con alguna manera de asegurar que una clase sí sobrescriba todos los métodos necesarios. La solución de Java a este problema es el *método abstracto*.

Un método abstracto se crea al especificar el modificador de tipo **abstract**. Un método abstracto no tiene cuerpo y, por lo tanto, no está implementado por la superclase. De ahí que una subclase debe sobrescribirlo (no es posible usar simplemente la versión definida en la superclase). Para declarar un método abstracto, se utiliza esta forma general:

abstract *tipo nombre*(*lista-parámetros*);

Como verá, no hay un cuerpo de método. El modificador **abstract** sólo puede usarse en métodos normales. No puede aplicarse ni a métodos **static** ni a constructores.

1. La sobrescritura de métodos ocurre cuando una subclase define un método que tiene la misma firma que un método en su superclase.
2. La sobrescritura de métodos permite a Java soportar el polimorfismo en tiempo de ejecución.
3. La versión de un método sobrescrito que se ejecuta está determinada por el tipo del objeto al que se hace referencia en el tiempo de la llamada. Por lo tanto, la determinación se hace en tiempo de ejecución.

Una clase que contiene uno o más métodos abstractos también debe declararse como abstracta al anteceder su declaración de **clase** con el especificador **abstract**. Debido a que una clase abstracta no define una implementación completa, es posible que no haya objetos de una clase abstracta. Por lo tanto, el intento de crear un objeto de una clase abstracta mediante el uso de **new** dará como resultado un error en tiempo de compilación.

Cuando una subclase hereda una clase abstracta, debe implementar todos los métodos abstractos en la superclase. Si no lo hace, entonces la subclase debe también especificarse como **abstract**. El atributo **abstract** se hereda entonces hasta el momento en que se alcanza una implementación completa.

Con el uso de una clase abstracta puede mejorar la clase **FormaDosD**. Debido a que no hay un concepto de área para la figura bidimensional indefinida, la siguiente versión del programa anterior declara **área( )** como **abstract** dentro de **FormaDosD**, y a **FormaDosD** como abstract. Esto, por supuesto, significa que todas las clases derivadas de **FormaDosD** deben sobrescribir a área().

```
// Crea una clase abstracta.
abstract class FormaDosD {  <-- Ahora FormaDosD es abstracta.
  private double ancho;
  private double alto;
  private String nombre;

  // Un constructor predeterminado.
  FormaDosD() {
    ancho = alto = 0.0;
    nombre = "null";
  }

  // Constructor con parámetros.
  FormaDosD(double w, double h, String n) {
    ancho = w;
    alto = h;
    nombre = n;
  }

  // Construye un objeto de ancho y alto iguales.
  FormaDosD(double x, String n) {
    ancho = alto = x;
    nombre = n;
  }

  // Construye un objeto a partir de un objeto.
  FormaDosD(FormaDosD ob) {
    ancho = ob.ancho;
    alto = ob.alto;
    nombre = ob.nombre;
  }
```

```
// Métodos de accesor para ancho y alto.
  double obtenerAncho() { return ancho; }
  double obtenerAlto() { return alto; }
  void establecerAncho(double w) { ancho = w; }
  void establecerAlto(double h) { alto = h; }

  String obtenerNombre() { return nombre; }

  void mostrarDim() {
    System.out.println("El ancho y alto son " +
                       ancho + " y " + alto);
  }

  // Ahora, área() es abstracta.
  abstract double área(); ←——— Convierte a área() en un método abstracto.
}

// Una subclase de FormaDosD para Triangs.
class Triang extends FormaDosD {
  private String estilo;

  // Un constructor predeterminado.
  Triang() {
    súper();
    estilo = "null";
  }

  // Constructor para Triang.
  Triang(String s, double w, double h) {
    súper(w, h, "un triángulo");

    estilo = s;
  }

  // Construye un triángulo isósceles.
  Triang(double x) {
    súper(x, "un triángulo"); // llama a un constructor de súperclase

    estilo = "isósceles";
  }

  // Construye un objeto a partir de un objeto.
  Triang(Triang ob) {
    súper(ob); // pasa un objeto al constructor de FormaDosD
    estilo = ob.estilo;
  }
```

```
  double área() {
    return obtenerAncho() * obtenerAlto() / 2;
  }

  void mostrarEstilo() {
    System.out.println("El triángulo es " + estilo);
  }
}

// Una subclase de FormaDosD para rectángulos.
class Rectang extends FormaDosD {
  // Un constructor predeterminado.
  Rectang() {
    súper();
  }

  // Constructor para rectángulo.
  Rectang(double w, double h) {
    super(w, h, "un rectángulo"); // llama a un constructor de súperclase
  }

  // Construye un cuadrado.
  Rectang(double x) {
    súper(x, "un rectángulo"); // llama a un constructor de superclase
  }

  // Construye un objeto a partir de un objeto.
  Rectang(Rectang ob) {
    súper(ob); // pasa un objeto a un constructor de FormaDosD
  }

  boolean esCuadrado() {
    if(obtenerAncho() == obtenerAlto()) return true;
    return false;
  }

  double área() {
    return obtenerAncho() * obtenerAlto();
  }
}

class FormaAbs {
  public static void main(String args[]) {
    FormaDosD formas[] = new FormaDosD[4];
```

```
    formas[0] = new Triang("recto", 8.0, 12.0);
    formas[1] = new Rectang(10);
    formas[2] = new Rectang(10, 4);
    formas[3] = new Triang(7.0);

    for(int i=0; i < formas.length; i++) {
      System.out.println("El objeto es " + formas[i].obtenerNombre());
      System.out.println("El área es " + formas[i].área());

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

Como se ilustra en el programa, todas las subclases de **FormaDosD** *deben* sobrescribir **área()**. Para probarlo usted mismo, cree una subclase que no sobrescriba **área()**. Recibirá un error en tiempo de compilación. Por supuesto, aún es posible crear una referencia a objeto de tipo **FormaDosD**, acción que el programa lleva a cabo. Sin embargo, ya no es posible declarar objetos de tipo **FormaDosD**. Debido a ello, en **main()**, la matriz formas puede acortarse a 4, y un objeto **FormaDosD** genérico ya no se crea.

Un último tema: observe que **FormaDosD** aún incluye los métodos **mostrarDim()** y **obtenerNombre()** y que éstos no son modificados por **abstract**. Es perfectamente aceptable (por supuesto muy común) que una clase abstracta contenga métodos concretos que la subclase tenga la libertad de usar como tales. Sólo aquellos métodos declarados como **abstract** deben ser sobrescritos por las subclases.

Comprobación de avance

1. ¿Qué es un método abstracto? ¿Cómo se crea?

2. ¿Qué es una clase abstracta?

3. ¿Puede un objeto de una clase abstracta convertirse en instancia?

1. Un método abstracto es aquel que no tiene un cuerpo. Consta de un tipo de regreso, un nombre y una lista de parámetros, y es precedido por la palabra clave **abstract**.
2. Una clase abstracta contiene por lo menos un método abstracto.
3. No.

HABILIDAD FUNDAMENTAL 7.10

Uso de final

Aunque la sobrescritura de métodos y la herencia son poderosas y útiles, en ocasiones deseará evitarlas. Por ejemplo, tal vez tenga una clase que encapsule el control de algún dispositivo de hardware. Más aún, esta clase podría ofrecer al usuario la capacidad de inicializar el dispositivo haciendo uso de información privada, es decir, de propietario. En este caso, no querrá que los usuarios de su clase sobrescriban el método de inicialización. Sin importar cuál sea la razón, en Java, si emplea la palabra clave **final**, es fácil evitar que un método sea sobrescrito o que una clase se herede.

Final evita la sobrescritura

Para evitar que un método sea sobrescrito, especifique **final** como modificador al inicio de su declaración. Los métodos declarados como **final** no pueden sobrescribirse. El siguiente fragmento ilustra **final**.

```
class A {
  final void met() {
    System.out.println("Éste es un método final.");
  }
}

class B extends A {
  void met() { // ERROR. No puede sobrescribirse.
    System.out.println("Illegal!");
  }
}
```

Debido a que **met()** está declarado como **final**, no puede sobrescribirse en **B**. Si trata de hacerlo, recibirá un error en tiempo de compilación.

Final evita la herencia

Puede evitar que una clase sea heredada si antecede su declaración con **final**. Al declarar una clase como **final**, se declara implícitamente que todos sus métodos serán también **final**. Como esperaría, es ilegal declarar una clase como **abstract** y **final** porque una clase abstracta está incompleta y depende de sus subclases para implementaciones completas.

He aquí un ejemplo de una clase **final**:

```
final class A {
  // ...
}
```

```
// La siguiente clase es ilegal.
Class B extends A {//ERROR! No puede ser subclase de A
  // ...
}
```

Como se desprende del comentario, resulta ilegal que **B** herede a **A** porque **A** está declarada como **final**.

Uso de final con miembros de datos

Además de los usos de **final** que se acaban de mostrar, también es posible aplicarlo a variables con el fin de crear lo que podría llamarse constantes con nombre. Si antecede el nombre de una variable de clase con **final**, su valor no podrá cambiarse durante todo el tiempo que el programa dure. Por supuesto, puede proporcionar a esa variable un valor inicial. Por ejemplo, en el módulo 6 se mostró una clase simple de administración de errores llamada **MsjError**. Esa clase correlacionaba una cadena legible con un código de error. Aquí, esa clase original se mejora con la adición de constantes **final** que definen los errores. Ahora, en lugar de que obtener**MsjError( )** pase un número como 2, puede pasar la constante de entero con nombre **ERRORDISC**.

```
// Regresa un objeto de cadena.
class MsjError {
  // Error al codificarse.
  final int ERRFUERA   = 0;
  final int ERRDENTRO  = 1; <-- Declara constantes final.
  final int ERRDISC    = 2;
  final int ERRÍNDICE  = 3;

  String msjs[] = {
    "Error de salida",
    "Error de entrada",
    "Disco lleno",
    "Índice fuera de límites"
  };

  // Regresa el mensaje de error.
  String obtenerMsjError(int i) {
    if(i >=0 & i < msjs.length)
      return msjs[i];
    else
      return "Código de error no válido";
  }
}
```

```
class FinalD {
  public static void main(String args[]) {
    MsjError err = new MsjError();

    System.out.println(err.obtenerMsjError(err.ERRFUERA));
    System.out.println(err.obtenerMsjError(err.ERRDISC));
  }
}
```

Uso de constantes **final**.

Observe cómo las constantes **final** se usan en **main()**. Debido a que son miembros de la clase **MsjError**, se deben acceder mediante un objeto de esa clase. Por supuesto, también las subclases pueden heredarlos y se puede acceder a ellos directamente dentro de esas subclases.

Como un tema de estilo, muchos programadores usan, como en el ejemplo anterior, identificadores en mayúsculas y minúsculas para las constantes **final**. Sin embargo, ésta no es una regla invariable.

Pregunte al experto

P: **¿Las variables** final **pueden hacerse** static**?**

R: Sí. Esto le permite hacer referencia a la constante a través de su nombre de clase en lugar de hacerlo a través de un objeto. Por ejemplo, si las constantes de **MsjError** se modifican mediante **static**, entonces las instrucciones **println()** en **main()** tendrán este aspecto:

```
System.out.println(err.obtenerMsjError(MsjError.ERRFUERA));
System.out.println(err.obtenerMsjError(MsjError.ERRDISC));
```

Comprobación de avance

1. ¿Cómo evita que un método se sobrescriba?
2. Si una clase se declara **final**, ¿puede heredarse?

1. Anteceda su declaración con la palabra clave **final**.
2. No.

HABILIDAD FUNDAMENTAL 7.11

La clase Object

Java define una clase especial llamada **Object** que es una superclase implícita de todas las demás clases. En otras palabras, todas las demás clases son subclases de **Object**. Esto significa que una variable de referencia de tipo **Object** puede hacer referencia a un objeto de cualquier otra clase. Además, debido a que las matrices están implementadas como clases, una variable de tipo **Object** puede también hacer referencia a cualquier matriz.

Object define los siguientes métodos, lo que significa que están disponibles para cualquier objeto.

Método	Propósito
Object clone()	Crea un nuevo objeto que es igual al objeto clonado.
boolean equals(Object *objeto*)	Determina si un objeto es igual a otro.
void finalize()	Se le llama antes de que un objeto que no se utiliza se recicle.
Class<? Extends Object> getClass	Obtiene la clase de un objeto en tiempo de ejecución.
int hashCode()	Regresa el código asociado con el objeto que invoca.
void notify()	Reanuda la ejecución de un subproceso que espera en el objeto que invoca.
void notifyAll()	Reanuda la ejecución de todos los subprocesos que esperan en el objeto que invoca.
String toString()	Regresa una cadena que describe el objeto.
void wait() void wait(long milisegundos) void wait(long *milisegundos,* int *nanosegundos*)	Espera en otro subproceso de ejecución.

Los métodos **getClass()**, **notify()**, **notifyAll()** y **wait()** se declaran como **final**. Usted puede sobrescribir los demás. Varios de estos métodos se describirán en las páginas posteriores de este libro. Sin embargo, por el momento examine estos dos métodos: **equals()** y **toString()**. El método **equals()** compara el contenido de dos objetos; devuelve **true** si son equivalentes, de lo contrario, devuelve **false**. El método **toString()** devuelve una cadena que contiene una descripción del objeto en el que se le llama. Además, se llama automáticamente a este método cuando se da salida a un objeto empleando **println()**. Muchas clases sobrescriben este método. Hacerlo así les permite desarrollar una descripción específica para los tipos de objetos que crean.

Un último detalle: observe la sintaxis inusual en el tipo de regreso de **getClass()**. Se trata de un tipo genérico. Los tipos genéricos constituyen una adición reciente (y con muchas opciones) a Java que permite que el tipo de datos usado por una clase o método se especifique como un parámetro. Los tipos genéricos se analizarán en el módulo 13.

Comprobación de dominio del módulo 7

1. ¿Una superclase tiene acceso a los miembros de una subclase? ¿Una subclase tiene acceso a los miembros de una superclase?
2. Cree una subclase de **FormaDosD** llamada **Círculo**. Incluya un método **área()** que calcule el área del círculo y un constructor que use super para inicializar la parte de **FormaDosD**.
3. ¿Cómo evita que una subclase tenga acceso a los miembros de una superclase?
4. Describa el objetivo y el uso de ambas versiones de súper.
5. Dada la siguiente jerarquía:

```
class Alfa { ...

class Beta extends Alfa { ...

class Gamma exteds Beta { ...
```

   ¿En qué orden se llama a los constructores de esas clases cuando se crea una instancia de un objeto de **Gamma**?
6. Una referencia a una superclase puede hacer referencia a un objeto de una subclase. Explique por qué esto es importante en relación con la sobrescritura de métodos.
7. ¿Qué es una clase abstracta?
8. ¿Cómo evita que un método se sobrescriba? ¿Cómo evita que una clase se herede?
9. Explique cómo se emplean la herencia, la sobrescritura de métodos y las clases abstractas para soportar el polimorfismo.
10. ¿Cuál clase es una superclase de todas las demás clases?
11. Una clase que contiene por lo menos un método abstracto debe, en sí misma, declararse como abstracta. ¿Cierto o falso?
12. ¿Cuáles palabras clave se usan para crear una constante con nombre?

Módulo 8

Paquetes e interfaces

HABILIDADES FUNDAMENTALES

En este módulo se examinan dos de las funciones más innovadoras de Java: los paquetes y las interfaces. Los *paquetes* son grupos de clases relacionadas. Ayudan a organizar su código y proporcionan otra capa de encapsulamiento. Una *interfaz* define un conjunto de métodos que serán implementados por una clase. Por sí sola, una interfaz no implementa ningún método, sino que es una construcción puramente lógica. Los paquetes y las interfaces le brindan un mayor control sobre la organización de su programa.

HABILIDAD FUNDAMENTAL 8.1

Paquetes

En programación agrupar partes relacionadas de un programa suele resultar útil. En Java, esto se logra con el uso de un paquete, el cual tiene dos propósitos. En primer lugar, proporciona un mecanismo mediante el cual piezas relacionadas de un programa pueden organizarse como una unidad. Las clases definidas dentro de un paquete deben ser accedidas mediante el nombre del paquete. Por consiguiente, un paquete proporciona una manera de denominar a una colección de clases. En segundo lugar, un paquete participa en el mecanismo de control de acceso de Java. Las clases definidas dentro de un paquete pueden hacerse privadas a ese paquete y no estar accesibles para el código que se encuentra fuera de éste. De modo que el paquete proporciona un medio mediante el cual las clases pueden encapsularse. Examinemos cada característica con más detenimiento.

En general, cuando asigna un nombre a una clase, usted está asignando un nombre a partir del *espacio de nombre*. Un espacio de nombre define una región declarativa. En Java, dos clases no pueden tener el mismo nombre del mismo espacio de nombre, así que, dentro de un nombre dado de espacio, cada nombre de clase debe ser único. Los ejemplos que se muestran en los módulos anteriores emplean el espacio de nombre predeterminado o global. Aunque esto es adecuado para programas cortos de ejemplo, se vuelve un problema a medida que los programas crecen y el espacio de nombre predeterminado se va sobrepoblando. En programas largos, llega a ser difícil encontrar nombres únicos para cada clase. Más aún, debe evitar colisiones de nombres con código creado por otros programadores que trabajan en el mismo proyecto, y con las bibliotecas de Java. La solución a este problema se encuentra en el paquete pues éste le ofrece una manera de dividir el espacio de nombre. Cuando se define una clase dentro de un paquete, el nombre de ese paquete está unido a cada clase, lo que evita las colisiones de nombres con otras clases que tienen el mismo nombre pero que están en otros paquetes.

Debido a que un paquete suele contener clases relacionadas, Java define derechos especiales de acceso al código dentro de un paquete. En un paquete, puede definir el código que es accesible para otro código dentro del mismo paquete, pero no para el que está fuera de éste. Esto le permite crear grupos independientes de clases relacionadas que mantienen su operación en privado.

Definición de un paquete

Todas las clases de Java pertenecen a algún **paquete**. Cuando no especifica una instrucción package, se usa el paquete predeterminado (o global). Más aún, el paquete predeterminado no tiene nombre, lo que lo vuelve transparente. Es por ello que hasta el momento no tenía que preocuparse por los

paquetes. Aunque el paquete predeterminado es adecuado para programas cortos de ejemplo, resulta inadecuado para aplicaciones reales. Casi siempre definirá uno o más paquetes para su código.

Para crear un **paquete**, debe poner un comando package en la parte superior del archivo fuente de Java. Las clases declaradas dentro de ese archivo pertenecerán entonces al paquete especificado. Debido a que un paquete define un espacio de nombre, los nombres de las clases que ponga en el archivo se volverán parte del espacio de nombre del paquete.

Ésta es la forma general de la instrucción **package**:

package *paquete*;

Aquí, *paquete* es el nombre del paquete. Por ejemplo, la siguiente instrucción crea un paquete llamado **Proyecto1**.

```
package Proyecto1;
```

Java usa el sistema de archivos para administrar los paquetes y almacena cada paquete en su propio directorio. Por ejemplo, los archivos **.class** de cualquier clase que declare como parte de **Proyecto1** deben almacenarse en un directorio llamado **Proyecto1**.

Como el resto de Java, los nombres de los paquetes son sensibles a las mayúsculas. Esto significa que el directorio en el que un paquete está almacenado debe tener precisamente el mismo nombre que el paquete. Si tiene problemas para probar los ejemplos de este módulo, recuerde revisar con cuidado los nombres de su paquete y su directorio.

Es posible incluir más de un archivo en la misma instrucción **package**. Esta instrucción simplemente especifica a cuál paquete pertenece la clase definida en el archivo. No excluye otras clases de otros archivos que son parte de ese mismo paquete. La mayor parte de los paquetes reales están dispersos en muchos archivos.

Puede crear una jerarquía de paquetes. Para ello, simplemente debe separar el nombre de cada paquete del anterior mediante el uso de un punto. La forma general de una instrucción package de varios niveles se muestra a continuación:

package *paq1.paq2.paq3...paqN;*

Por supuesto, debe crear directorios que soporten la jerarquía de paquetes que forme. Por ejemplo,

```
Paquete X.Y.Z;
```

debe estar almacenado en …/X/Y/Z, donde ... especifica la ruta hacia los directorios especificados.

Búsqueda de paquetes y CLASSPATH

Como se acaba de explicar, los paquetes tienen su reflejo en los directorios. Esto plantea una importante pregunta: ¿cómo sabe el sistema en tiempo de ejecución de Java dónde buscar los paquetes que crea? La respuesta tiene dos vertientes: en primer lugar, como opción predeterminada, el sistema de Java en tiempo de ejecución usa el directorio de trabajo actual como punto de partida. Por lo tanto, si sus archivos de clase son el directorio actual o un subdirectorio del directorio actual, Java los encontrará. En segundo lugar, puede especificar una o varias rutas de directorio si define la variable de entorno **CLASSPATH**.

Por ejemplo, considere la siguiente especificación de paquete:

```
package MiPaquete;
```

Con el objetivo de que un programa encuentre **MiPaquete**, una de las siguientes aseveraciones debe ser cierta: el programa se ejecuta desde un directorio inmediatamente superior a **MiPaquete**, o se ha definido **CLASSPATH** para que incluya la ruta a **MiPaquete**. La primera opción es la más fácil (y no requiere un cambio a **CLASSPATH**), pero la segunda opción le permite encontrar **MiPaquete**, sin importar en cuál directorio se encuentra el programa. Al final de cuentas, la decisión es suya.

Una última consideración: para evitar una confusión, es mejor mantener todos los archivos **.java** y **.class** asociados con paquetes en sus propios directorios de paquetes.

El efecto preciso y la configuración de **CLASSPATH** han cambiado con el tiempo y con cada revisión de Java. Se recomienda que vaya al sitio Web de Sun **java.sun.com** para conocer la información más reciente.

Un ejemplo corto de paquete

Teniendo el análisis anterior en cuenta, pruebe este paquete corto de ejemplo. Cree una base simple de datos de libros que se encuentre dentro de un paquete llamado **PaqueteLibro**.

```
// Un paquete corto de demostración.
package PaqueteLibro;

class Libro {
  private String título;
  private String autor;
  private int fechaPubl;

  Libro(String t, String a, int d) {
```

Este archivo es parte del paquete **PaqueteLibro**.

Por tanto, **Libro** es parte de **PaqueteLibro**.

```
    título = t;
    autor = a;
    fechaPubl = d;
  }

  void mostrar() {
    System.out.println(título);
    System.out.println(autor);
    System.out.println(fechaPubl);
    System.out.println();
  }
}

class LibroDemo {
  public static void main(String args[]) {
    Libro libros[] = new Libro[5];

    libros[0] = new Libro("Principios de Java",
                          "Schildt", 2005);
    libros[1] = new Libro("Java: The Complete Reference",
                          "Schildt", 2005);
    libros[2] = new Libro("The Art of Java",
                          "Schildt y Holmes", 2003);
    libros[3] = new Libro("Tormenta roja",
                           "Clancy", 1986);
    libros[4] = new Libro("En el camino",
                          "Kerouac", 1955);

    for(int i=0; i < libros.length; i++) libros[i].mostrar();
  }
}
```

LibroDemo también es parte de **PaqueteLibro**.

Llame a este archivo **LibroDemo.java** y póngalo en un directorio llamado **LibroPaquete**.

A continuación, compile el archivo. Asegúrese de que el archivo **.class** resultante también esté en el directorio **LibroPaquete**. Luego trate de ejecutar la clase empleando la siguiente línea de comandos:

```
java PaqueteLibro.LibroDemo
```

Recuerde que usted tendrá que estar en el directorio superior **PaqueteLibro** cuando ejecute este programa o haber establecido de manera adecuada la variable de entorno **CLASSPATH**.

Como se explicó, ahora **LibroDemo** y **Libro** son parte del paquete **PaqueteLibro**. Esto significa que **LibroDemo** no puede ejecutarse por sí solo. Es decir, no puede usar esta línea de comandos:

```
java LibroDemo
```

En cambio, **LibroDemo** debe calificarse con su nombre de paquete.

Comprobación de avance

1. ¿Qué es un paquete?
2. Muestre cómo declarar un paquete llamado **PaqueteHerramienta**.
3. ¿Qué es **CLASSPATH**?

HABILIDAD FUNDAMENTAL 8.2

Acceso a paquetes y miembros

En módulos anteriores se han presentado los fundamentos del control de acceso, incluidos los especificadores **private** y **public**; sin embargo, éstos no lo han demostrado todo. La razón es que los paquetes también participan en el mecanismo de control de acceso. De ahí que un análisis completo haya tenido que permanecer en espera hasta que los paquetes sean estudiados.

La visibilidad de un elemento está determinada por su especificación de acceso (**private**, **public**, **protected** o predeterminado) y el paquete en el que reside. Así, la visibilidad de un elemento está determinada por su visibilidad dentro de una clase y dentro de un paquete. Este método de varias capas para el control de acceso soporta una serie variada de privilegios de acceso. En la tabla 8.1 se presenta un resumen de los diversos niveles de acceso. Examinemos de manera individual cada opción de acceso.

Si un miembro de una clase no tiene un especificador de acceso específico, entonces es visible dentro de su paquete pero no fuera de él. Por lo tanto, usará la especificación de acceso predeterminado para elementos que quiera mantener privados para un paquete pero públicos dentro de ese paquete.

Los miembros declarados específicamente como **public** son visibles en todos lados, incluso en diferentes clases y distintos paquetes. No hay restricción para su uso o acceso.

Un miembro **private** sólo es accesible para los demás miembros de su clase. Un miembro **private** no se ve afectado por su membresía a un paquete.

Un miembro especificado como **protected** es accesible dentro de su paquete y para todas las subclases, incluidas las de otros paquetes.

La tabla 8.1 sólo se aplica a los miembros de las clases. Una clase sólo tiene dos niveles posibles de acceso: predeterminado y public. Cuando una clase se declara **public**, es accesible para cualquier otro código. Si una clase tiene acceso predeterminado, sólo puede acceder a ella el resto del código del mismo paquete. Además, una clase declarada como **public** debe residir en un archivo con el mismo nombre.

1. Un paquete es un contenedor de clases. Desempeña una función de organización y encapsulamiento.
2. `package PaqueteHerramienta;`
3. **CLASSPATH** es la variable de entorno que especifica la ruta hacia las clases.

	Miembro privado	Miembro predeterminado	Miembro protegido	Miembro público
Visible dentro de la misma clase	Sí	Sí	Sí	Sí
Visible para la subclase dentro del mismo paquete	No	Sí	Sí	Sí
Visible para el código que no es subclase dentro del mismo paquete	No	Sí	Sí	Sí
Visible para la subclase dentro de un paquete diferente	No	No	Sí	Sí
Visible para el código que no es subclase dentro de un paquete diferente	No	No	No	Sí

Tabla 8.1 Acceso a miembros de clase

Comprobación de avance

1. Si un miembro de clase tiene acceso predeterminado dentro de un paquete, ¿ese miembro es accesible para otros paquetes?
2. ¿Qué hace **protected**?
3. Una subclase dentro de sus paquetes puede acceder a un miembro **private**. ¿Cierto o falso?

Un ejemplo de acceso a paquete

En el ejemplo de **package** que se mostró antes, **Libro** y **LibroDemo** estaban en el mismo paquete, de modo que no había problema en que **LibroDemo** usara **Libro**, porque el privilegio de acceso predeterminado otorga acceso a todos los miembros del mismo paquete. Sin embargo, si **Libro**

1. No.
2. Permite que un miembro sea accesible para un código adicional en su paquete y para todas las subclases, sin importar en cuál paquete se encuentre la subclase.
3. Falso.

estuviera en el mismo paquete y **LibroDemo** estuviera en otro, la situación sería diferente. En este caso, se negaría el acceso a **Libro**. Para que **Libro** esté disponible para otros paquetes, debe llevar a cabo tres cambios. En primer lugar, es necesario que **Libro** sea declarado como **public**. Esto hace que **Libro** sea visible fuera de **PaqueteLibro**. En segundo lugar, debe hacer **public** a su constructor. Finalmente, el método **mostrar( )** debe ser **public**. Esto les permite ser visibles también fuera de **PaqueteLibro**. Así que, para que otros paquetes puedan usar **Libro**, éste debe recodificarse de la siguiente manera:

```
// Libro recodificado para acceso public.
package PaqueteLibro;

public class Libro {
  private String título;
  private String autor;
  private int fechaPubl;

  // Ahora es public.
  public Libro(String t, String a, int d) {
    título = t;
    autor = a;
    fechaPubl = d;
  }

  // Ahora es public.
  public void mostrar() {
    System.out.println(título);
    System.out.println(autor);
    System.out.println(fechaPubl);
    System.out.println();
  }
}
```

Libro y sus miembros deben ser **public** para que otros paquetes los usen.

Para usar **Libro** desde otro paquete, debe usar la instrucción **import** que se describirá en la siguiente sección, o debe calificar completamente su nombre para incluir su especificación completa de paquete. Por ejemplo, he aquí una clase llamada **UsarLibro** que se encuentra dentro del paquete **PaqueteLibroB**. Esta clase califica por completo a **Libro** para emplearlo.

```
// Esta clase está en el paquete PaqueteLibroB.
package PaqueteLibroB;

// Usa la clase Libro de PaqueteLibro.
class UseLibro {
  public static void main(String args[]) {
    PaqueteLibro.Libro libros[] = new PaqueteLibro.Libro[5];

    libros[0] = new PaqueteLibro.Libro("Principios de Java",
```

Califica **Libro** con su nombre de paquete: **PaqueteLibro**.

```
                        "Schildt", 2005);
    libros[1] = new PaqueteLibro.Libro("Java: The Complete Reference",
                        "Schildt", 2005);
    libros[2] = new PaqueteLibro.Libro("The Art of Java",
                        "Schildt y Holmes", 2003);
    libros[3] = new PaqueteLibro.Libro("Tormenta Roja",
                         "Clancy", 1986);
    libros[4] = new PaqueteLibro.Libro("En el camino",
                        "Kerouac", 1955);

    for(int i=0; i < libros.length; i++) libros[i].mostrar();
  }
}
```

Observe cómo cada uso de **Libro** está antecedido por el calificador **PaqueteLIbro**. Sin esta especificación, no encontraría **Libro** al tratar de compilar **UsarLibro**.

HABILIDAD FUNDAMENTAL 8.3

Los miembros protegidos

Las personas que por primera vez utilizan Java a veces se sienten confundidos por el significado y el uso de **protected**. Como se explicó, el especificador **protected** crea un miembro que es accesible dentro de su paquete y para las subclases de otros paquetes. Por lo tanto, un miembro **protected** está disponible para todas las subclases pero está aún protegido contra el acceso arbitrario por parte de un código fuera de su paquete.

Para comprender mejor los efectos de **protected**, revisemos un ejemplo. En primer lugar, cambie la clase **Libro** para que sus variables de instancia sean **protected**, como se muestra a continuación.

```
// Haga que esta variable de instancia en Libro sea protected.
package PaqueteLibro;

public class Libro {
  // these are now protected
  protected String título;
  protected String autor;
  protected int fechaPubl;

  public Libro(String t, String a, int d) {
    título = t;
    autor = a;
    fechaPubl = d;
  }
```

Ahora son **protected**.

```
   public void mostrar() {
    System.out.println(título);
    System.out.println(autor);
    System.out.println(fechaPubl);
    System.out.println();
  }
}
```

A continuación, cree una subclase de **Libro** llamada **ExtLibro**, y una clase denominada **ProtectDemo** que utilice **ExtLibro**. **ExtLibro** agrega un campo que almacena el nombre del editor y varios métodos de accesor. Estas dos clases estarán en su propio paquete llamado **PaqueteLibroB**. Aquí se muestran.

```
// Demuestra Protected.
package PaqueteLibroB;

class ExtLibro extends PaqueteLibro.Libro {
  private String editor;

  public ExtLibro(String t, String a, int d, String p) {
    super(t, a, d);
    editor = p;
  }

  public void mostrar() {
    super.mostrar();
    System.out.println(editor);
    System.out.println();
  }

  public String obtenerEditor() { return editor; }
  public void establecerEditor(String p) { editor = p; }

  /* Esto es correcto porque la subclase puede
     accesar un miembro protegido. */
  public String obtenerTítulo() { return título; }
  public void estaclecerTítulo(String t) { título = t; }
  public String obtenerAutor() { return autor; }
  public void establecerAutor(String a) { autor = a; }
  public int obtenerFechaPubl() { return fechaPubl; }
  public void establecerFechaPubl(int d) { fechaPubl = d; }
}
```

Está permitido el acceso a los miembros de **Libro** para la subclase.

```
class ProtectDemo {
  public static void main(String args[]) {
    ExtLibro libros[] = new ExtLibro[5];

    libros[0] = new ExtLibro("Principios de Java",
                        "Schildt", 2005, "Osborne "McGraw-Hill");
    libros[1] = new ExtLibro("Java: The Complete Reference",
                        "Schildt", 2005, "Osborne/McGraw-Hill");
    libros[2] = new ExtLibro("The Art of Java",
                        "Schildt y Holmes", 2003,
                        "Osborne/McGraw-Hill");
    libros[3] = new ExtLibro("Tormenta Roja",
                        "Clancy", 1986, "Putnam");
    libros[4] = new ExtLibro("En el camino",
                        "Kerouac", 1955, "Viking");

    for(int i=0; i < libros.length; i++) libros[i].mostrar();

    // encuentra libros por autor
    System.out.println("Se muestran todos los libros de Schildt.");
    for(int i=0; i < libros.length; i++)
      if(libros[i].obtenerAutor() == "Schildt")
        System.out.println(libros[i].obtenerTítulo());

//    libros[0].título = "probar título"; // Error: no es accesible
  }
}
```

Acceso al campo **protected** no está permitido por miembros que no son de la subclase.

Primero observe el código dentro de **ExtLibro**. Debido a que **ExtLibro** extiende **Libro**, tiene acceso a los miembros **protected** de **Libro**, aunque **ExtLibro** esté en un paquete diferente. De ahí que **ExtLibro** pueda acceder a **título**, **autor** y **fechaPubl** directamente como lo hace en los métodos de accesor que crea para esas variables. Sin embargo, en **ProtectDemo**, se niega el acceso a esas variables porque **ProtectDemo** no es una subclase de **Libro**. Por ejemplo, si elimina el símbolo de comentario de la siguiente línea, el programa no se compilará:

```
//    libros[0].título = "probar título"; // Error: no es accesible
```

HABILIDAD FUNDAMENTAL 8.4

Importación de paquetes

Al emplear una clase de otro paquete, puede calificar completamente el nombre de la clase con el nombre de su paquete, como en el ejemplo anterior. Sin embargo, este método podría volverse tedioso y molesto muy rápidamente, sobre todo si las clases que está calificando se encuentran profundamente

Pregunte al experto

P: Sé que C++ también incluye un especificador de acceso llamado protected. ¿Es similar al de Java?

R: Similar pero no igual. En C++, protected crea un miembro que al que puede accederse desde la subclase pero que de otra manera es private. En Java, protected crea un miembro que puede ser accedido por cualquier código dentro de su paquete pero sólo puede ser accedido por las subclases fuera de su paquete. Es necesario que tenga cuidado ante esta diferencia cuando mueva código entre C++ y Java.

anidadas en una jerarquía de paquetes. Como Java fue inventado por programadores para programadores (y programadores a los que no les gustan los constructores tediosos) no es sorprendente que exista un método más eficiente para utilizar el contenido de los paquetes: la instrucción **import**. Con el uso de **import** puede traer a la vista uno o más miembros de un paquete. Esto le permite usar estos miembros directamente, sin una calificación explícita de paquete.

He aquí la forma general de la instrucción **import**:

import *paquete.nombreclase*;

Aquí, *paquete* es el nombre del paquete, el cual puede incluir su ruta completa, y *nombreclase* es el nombre de la clase que se está importando. Si quiere importar todo el contenido de un paquete, use un asterisco (*) para el nombre de la clase. He aquí ejemplos de ambas formas:

```
import MiPaquete.MiClase
import MiPaquete.*;
```

En el primer caso, la clase **MiClase** se importa desde **MiPaquete**. En el segundo, todas las clases de **MiPaquete** se importan. En un archivo fuente de Java, las instrucciones **import** ocurren inmediatamente después de la instrucción **package** (si existe) y antes de cualquier definición de clase.

Puede usar **import** para traer a la vista el paquete **PaqueteLibro** con el fin de que la clase **Libro** pueda emplearse sin calificación. Para ello, simplemente agregue esta instrucción **import** en la parte superior de cualquier archivo que utilice **Libro**.

```
import PaqueteLibro.*;
```

Por ejemplo, he aquí la clase **UseLibro** que está registrada para usar **import**.

```
// Demuestra import.
package PaqueteLibroB;
import PaqueteLibro.*;

// Usa la clase Libro de PaqueteLibro.
class UseLibro {
  public static void main(String args[]) {
    Libro libros[] = new Libro[5];

    libros[0] = new Libro("Principios de Java",
                          "Schildt", 2005);
    libros[1] = new Libro("Java: The Complete Reference",
                          "Schildt", 2005);
    libros[2] = new Libro("The Art of Java",
                          "Schildt y Holmes", 2003);
    libros[3] = new Libro("Tormenta Roja",
                           "Clancy", 1986);
    libros[4] = new Libro("On the Road",
                          "Kerouac", 1955);

    for(int i=0; i < libros.length; i++) libros[i].mostrar();
  }
}
```

Importa **PaqueteLibro**.

Ahora puede hacer referencia a **Libro** directamente, sin calificación.

Observe que ya no necesita calificar **Libro** con su nombre de paquete.

Pregunte al experto

P: **¿La importación de un paquete tiene impacto en el desempeño de mi programa?**

R: ¡Sí y no! La importación de un paquete puede crear una pequeña cantidad de carga adicional durante la compilación, pero no tiene impacto en el desempeño en tiempo de ejecución.

HABILIDAD FUNDAMENTAL 8.5

Las bibliotecas de clases de Java se encuentran en paquetes

Como se explicó en páginas anteriores, Java define un gran número de clases estándar que están disponibles para todos los programas. A esta biblioteca de clases se le conoce por lo general como la API (Application Programming Interface, interfaz de programación de aplicaciones). La API de Java está almacenada en paquetes. En la parte superior de la jerarquía de paquetes se encuentra **java**. De ésta se desprenden varios subpaquetes, incluidos los siguientes:

Subpaquete	Descripción
java.lang	Contiene un gran número de clases de propósito general
java.io	Contiene las clases de E/S
java.net	Contiene las clases que soportan trabajo en red
java.applet	Contiene clases para crear applets
java.awt	Contiene clases que soportan el juego de herramientas abstractas de Windows (Abstract Windows Toolkit)

Desde el principio de este libro, se ha estado empleando **java.lang**. Éste contiene, entre otras, la clase **System** que ya ha utilizado cuando realiza salida empleando **println()**. El paquete **java.lang** es único porque se importa automáticamente en todos los programas de Java. Es por ello que no ha tenido que importar **java.lang** en los programas del ejemplo anterior. Sin embargo, debe importar específicamente los demás paquetes. En los módulos siguientes examinaremos varios paquetes.

Comprobación de avance

1. ¿Cómo incluye otro paquete en un archivo fuente?
2. Muestre la manera de incluir todas las clases en un paquete llamado **PaqueteHerramienta**.
3. ¿Es necesario que incluya explícitamente **java.lang**?

1. Utilice la instrucción **import**.
2. `import PaqueteHerramienta.*;`
3. No.

HABILIDAD FUNDAMENTAL 8.6

Interfaces

En la programación orientada a objetos, suele resultar útil definir lo que una clase debe hacer pero no la forma en que procederá. Ya ha visto un ejemplo de ello: el método abstract. Un método abstracto define la firma de un método, pero no proporciona implementación. Una subclase debe proporcionar su propia implementación de cada método abstracto definido por su superclase. Por lo tanto, un método abstracto especifica la *interfaz* al método pero no la *implementación*. Aunque las clases y los métodos abstractos son útiles, es posible llevar este concepto un paso más allá. En Java, puede separar por completo una interfaz de clase de su implementación usando la palabra clave **interface**.

Las interfaces son sintácticamente similares a las clases abstractas. Sin embargo, en una interfaz ningún método puede incluir un cuerpo. Es decir, una interfaz no proporciona ningún tipo de implementación. Especifica lo que debe hacerse, pero no el cómo. Una vez que se ha definido una interfaz, cualquier cantidad de clases pueden implementarla. Además, una clase puede implementar cualquier cantidad de interfaces.

Para implementar una interfaz, una clase debe proporcionar cuerpos (implementaciones) para los métodos descritos por la interfaz. Cada clase tiene la libertad de determinar los detalles de su propia implementación. De modo que dos clases podrían implementar la misma interfaz de diferentes maneras, pero cada clase soporta aún el mismo conjunto de métodos. Así, el código que conoce a la interfaz puede emplear objetos de cualquier clase porque la interfaz de esos objetos es la misma. Al proporcionar la palabra clave **interface**, Java le permite utilizar completamente el aspecto "una interfaz, varios métodos" del polimorfismo.

He aquí la forma general de una interfaz:

```
acceso interface nombre {
  tipo-ret nombre-método1(lista-pams);
  tipo-ret nombre-método2(lista-pams);
  tipo var1 = valor;
  tipo var2 = valor;
  // ...
  tipo-ret nombre-métodoN(lista-pams);
  tipo varN = valor;
}
```

Aquí, *acceso* es **public** o no se emplea. Cuando se incluye un especificador de acceso, se obtiene el acceso predeterminado, y la interfaz sólo está disponible para los demás miembros de su paquete. Cuando ésta se declara **public**, cualquier otro código puede usarla. (Cuando una **interfaz** se declara **public**, debe estar en un archivo del mismo nombre.) *nombre* es el nombre de la interfaz y puede ser cualquier identificador válido.

Los métodos se declaran usando sólo su tipo de regreso y su forma. Son, en esencia, métodos abstractos. Como se explicó, en una **interfaz** ningún método puede tener una implementación. Por lo

tanto, cada clase que incluya una **interfaz** debe implementar todos los métodos. En una interfaz, los métodos son implícitamente **public**.

Las variables declaradas en una **interface** no son variables de instancia. En cambio, son implícitamente **public**, **final** y **static** y deben inicializarse. Son, por consiguiente, esencialmente constantes.

He aquí un ejemplo de una definición de **interface**, la cual especifica la interfaz a una clase que genera una serie de números.

```
public interface Serie {
  int obtenerSiguiente(); // regresa el siguiente número de la serie
  void restablecer(); // reinicia
  void establecerInicio(int x); // establece el valor inicial
}
```

Esta interfaz se declara **public** para que el código de cualquier paquete la pueda implementar.

HABILIDAD FUNDAMENTAL 8.7

Implementación de interfaces

Una vez que se ha definido una **interfaz**, una o más clases pueden implementarla. Para implementar una interfaz, debe incluir la cláusula **implements** en una definición de clase y luego crear los métodos definidos por la interfaz. La forma general de una clase que incluye la cláusula **implements** tiene este aspecto:

acceso class *nombreclase* extends *superclase* implements *interfaz* {
 // cuerpo-clase
}

Aquí, *acceso* es **public** o no se utiliza. La cláusula **extends** es, por supuesto, opcional. Para implementar más de una interfaz, las interfaces se separan mediante comas.

Los métodos que implementan una interfaz deben declararse **public**. Además, la firma de tipo del método de implementación debe coincidir exactamente con la firma de tipo especificada en la definición de **interface**.

He aquí un ejemplo que implementa la interfaz **Serie** que se mostró antes. En él se crea una clase llamada **PorDos** que genera una serie de números que aumenta de dos en dos.

```
// Implementa Serie.
class PorDos implements Serie {
  int inicio;
  int val;

  PorDos() {
    inicio = 0;
```

Implementa la interfaz serie

```
    val = 0;
  }

  public int obtenerSiguiente() {
    val += 2;
    return val;
  }

  public void restablecer() {
    inicio = 0;
    val = 0;
  }

  public void establecerInicio(int x) {
    inicio = x;
    val = x;
  }
}
```

Observe que los métodos **obtenerSiguiente()**, **restablecer()** y **establecerInicio()** se declaran usando el especificador de acceso **public**, lo cual es necesario. Cada vez que implementa un método definido por una interfaz, éste debe implementarse como **public** porque todos los miembros de una interfaz son implícitamente **public**.

He aquí una clase que demuestra **PorDos**.

```
class SerieDemo {
  public static void main(String args[]) {
    PorDos ob = new PorDos();

    for(int i=0; i < 5; i++)
      System.out.println("El siguiente valor es " +
                         ob.obtenerSiguiente());

    System.out.println("\nRestableciendo");
    ob.restablecer();
    for(int i=0; i < 5; i++)
      System.out.println("El siguiente valor es " +
                         ob.obtenerSiguiente());

    System.out.println("\nEmpezando en 100");
    ob.establecerInicio(100);
    for(int i=0; i < 5; i++)
      System.out.println("El siguiente valor es " +
                         ob.obtenerSiguiente());
  }
}
```

Ésta es la salida de este programa:

```
El siguiente valor es 2
El siguiente valor es 4
El siguiente valor es 6
El siguiente valor es 8
El siguiente valor es 10

Restableciendo
El siguiente valor es 2
El siguiente valor es 4
El siguiente valor es 6
El siguiente valor es 8
El siguiente valor es 10

Empezando en 100
El siguiente valor es 102
El siguiente valor es 104
El siguiente valor es 106
El siguiente valor es 108
El siguiente valor es 110
```

Un hecho común y permitido es que las clases que implementan interfaces definan miembros adicionales por su cuenta. Por ejemplo, la siguiente versión de **PorDos** agrega el método **obtenerAnterior()**, que regresa el valor anterior.

```
// Implementa Serie y agrega obtenerAnterior().
class PorDos implements Serie {
  int inicio;
  int val;
  int ant;

  PorDos() {
    inicio = 0;
    val = 0;
    ant = -2;
  }

  public int obtenerSiguiente() {
    ant = val;
    val += 2;
    return val;
  }

  public void restablecer() {
    inicio = 0;
```

```
    val = 0;
    ant = -2;
  }

  public void establecerInicio(int x) {
    inicio = x;
    val = x;
    ant = x - 2;
  }

  int obtenerAnterior() {          <-- Agrega un método no definido por Serie.
    return ant;
  }
}
```

Observe que la adición de **obtenerAnterior()** requirió un cambio a las implementaciones de los métodos definidos por **Serie**. Sin embargo, debido a que la interfaz de esos métodos permanece igual, el cambio no es notorio y no rompe con el código anterior. Esa es una de las ventajas de las interfaces.

Como ya se explicó, cualquier cantidad de clases pueden implementar una **interfaz**. Por ejemplo, he aquí una clase llamada **PorTres** que genera una serie que consta de múltiplos de tres.

```
// Implementa Serie.
class PorTres implements Serie {   <-- Implementa Serie de una manera diferente.
  int inicio;
  int val;

  PorTres() {
    inicio = 0;
    val = 0;
  }

  public int obtenerSiguiente() {
    val += 3;
    return val;
  }

  public void restablecer() {
    inicio = 0;
    val = 0;
  }

  public void establecerInicio(int x) {
    inicio = x;
    val = x;
  }
}
```

Un comentario adicional: si una clase incluye una interfaz pero no implementa por completo los métodos definidos por esa interfaz, entonces dicha clase deberá declararse como **abstract**. No pueden crearse objetos de esa clase, pero ésta puede usarse como una superclase abstracta, lo que permite que las subclases proporcionen la implementación completa.

HABILIDAD FUNDAMENTAL 8.8

Uso de referencias a interfaces

Tal vez le sorprenda un poco saber que puede declarar una variable de referencia de un tipo de interfaz. En otras palabras, puede crear una variable de referencia a interfaz. Este tipo de variable puede hacer referencia a cualquier objeto que implemente su interfaz. Cuando usted llama a un método sobre un objeto mediante una referencia a interfaz, se ejecuta la versión del método implementada por el objeto. Este proceso es similar al uso de una referencia a superclase para acceder a un objeto de una subclase, como ya se describió en el módulo 7.

El siguiente ejemplo ilustra este proceso pues utiliza la misma variable de referencia a interfaz para llamar a métodos sobre objetos de **PorDos** y **PorTres**.

```
// Demuestra las referencias a interfaz.

class PorDos implements Serie {
  int inicio;
  int val;

  PorDos() {
    inicio = 0;
    val = 0;
  }

  public int obtenerSiguiente() {
    val += 2;
    return val;
  }

  public void restablecer() {
    inicio = 0;
    val = 0;
  }

  public void establecerInicio(int x) {
    inicio = x;
    val = x;
  }
}
```

```
class PorTres implements Serie {
  int inicio;
  int val;

  PorTres() {
    inicio = 0;
    val = 0;
  }

  public int obtenerSiguiente() {
    val += 3;
    return val;
  }

  public void restablecer() {
    inicio = 0;
    val = 0;
  }

  public void establecerInicio(int x) {
    inicio = x;
    val = x;
  }
}

class SerieDemo2 {
  public static void main(String args[]) {
    PorDos dosOb = new PorDos();
    PorTres tresOb = new PorTres();
    Serie ob;

    for(int i=0; i < 5; i++) {
      ob = dosOb;
      System.out.println("El siguiente valor de PorDos es " +
                         ob.obtenerSiguiente());
      ob = tresOb;
      System.out.println("El siguiente valor de PorTres es " +
                         ob.obtenerSiguiente());
    }
  }
}
```

Accesa un objeto vía una referencia a interfaz.

En **main()**, **Object** se declara para que sea una referencia a una interfaz **Serie**. Esto significa que puede usarse para almacenar referencias a cualquier objeto que implemente **Serie**. En este caso, se usa para hacer referencia a **dosOb** y **tresOb**, que son objetos de tipo **PorDos** y **PorTres**, respectivamente,

los cuales implementan **Serie**. Una variable de referencia a interfaz sólo conoce los métodos declarados por su declaración **interface**. Por lo tanto, **Object** no podría usarse para acceder a ninguna otra variable o método que pudiera estar soportado por el objeto.

Comprobación de avance

1. ¿Qué es una interfaz? ¿Qué palabra clave se utiliza para definir una?
2. ¿Para qué sirve **implements**?
3. ¿Una variable de referencia a interfaz puede hacer referencia a un objeto que implemente dicha interfaz?

Proyecto 8.1 Creación de una interfaz de cola

ICCar.java
ICDemo.java

Para ver la capacidad de las interfaces en acción, revisaremos un ejemplo práctico. En módulos anteriores, desarrolló una clase llamada **Cola** que implementaba una cola simple de tamaño fijo para caracteres. Sin embargo, existen muchas maneras de implementar una cola. Por ejemplo, la cola puede ser de tamaño fijo o "creciente". La cola puede ser lineal, en cuyo caso puede usarse hacia arriba, o circular, en cuyo caso los elementos pueden entrar a medida que vayan saliendo. La cola también puede mantenerse en una matriz, una lista vinculada, un árbol binario, etc. No importa cómo se implemente la cola, la interfaz a la cola permanece igual, y los métodos **obtener( )** y **colocar( )** definen la interfaz a la cola de manera independiente a los detalles de la implementación. Debido a que la interfaz a una cola se encuentra separada de su implementación, resulta fácil definir una interfaz de cola dejando a cada implementación que defina los detalles.

En este proyecto, creará una interfaz para una cola de caracteres y tres implementaciones. Estas tres implementaciones usarán una matriz para almacenar los caracteres. Una de las colas será de tamaño fijo y lineal como la que se desarrolló antes, mientras que otra será circular. En una cola circular, cuando se llega al final de la matriz, los índices **obtener** y **colocar** regresan automáticamente el bucle al inicio. Por lo tanto, es posible almacenar cualquier cantidad de elementos, siempre y cuando se vayan desprendiendo elementos. La implementación final crea una cola dinámica, que crece lo necesario cuando su tamaño es excedido.

1. Una interfaz define los métodos que una clase debe implementar pero no define una implementación por su cuenta. Está definida por la palabra clave **interface**.
2. Para implementar una interfaz, inclúyala en una clase empleando la palabra clave **implements**.
3. Sí.

Paso a paso

1. Cree un archivo llamado **ICCar.java** y ponga en el archivo la siguiente definición de interfaz.

```
// Una interfaz de cola de caracteres.
public interface ICCar {
  // Coloca un carácter en la cola.
  void colocar(char ch);

  // obtiene un carácter de la cola.
  char obtener();
}
```

   Como puede ver, esta información es muy simple y sólo cuenta con dos métodos. Cada clase que implemente **ICCar** necesitará implementar estos métodos.

2. Cree un archivo llamado **ICDemo.java**.

3. Empiece la creación de **ICDemo.java** agregando la clase **ColaFija** que se muestra a continuación:

```
// Clase de cola de caracteres de tamaño fijo.
class ColaFija implements ICCar {
  private char q[]; // esta matriz contiene la cola
  private int colocarlug, obtenerlug; // los índices colocar y obtener

  // Construye una cola vacía dado su tamaño.
  public ColaFija(int dim) {
    q = new char[dim+1]; // asigna memoria para la cola
    colocarlug = obtenerlug = 0;
  }

  // Coloca un carácter en la cola.
  public void colocar(char ch) {
    if(colocarlug==q.length-1) {
      System.out.println(" - La cola se ha llenado.");
      return;
    }

    colocarlug++;
    q[colocarlug] = ch;
  }

  // obtiene un carácter de la cola.
  public char obtener() {
    if(obtenerlug == colocarlug) {
      System.out.println(" - La cola se encuentra vacía.");
      return (char) 0;
```

(continúa)

```
    }
    obtenerlug++;
    return q[obtenerlug];
  }
}
```

Esta implementación de **ICCar** está adaptada a partir de la clase **Cola** que se mostró en el módulo 5 y que ya debe resultarle familiar.

4. Agregue la clase **ColaCircular** que se muestra aquí a **ICDemo.java**. Implementa una cola circular a caracteres.

```
// Una cola circular.
class ColaCircular implements ICCar {
  private char q[]; // esta matriz contiene la cola
  private int colocarlug, obtenerlug; // los índices colocar y obtener

  // Construye una cola vacía dado su tamaño.
  public ColaCircular(int dim) {
    q = new char[dim+1]; // asigna memoria para la cola
    colocarlug = obtenerlug = 0;
  }

  // Coloca un carácter en la cola.
  public void colocar(char ch) {
    /* La cola está llena si colocarlug es menor en uno que
       obtenerlug, o si colocarlug está al final de la matriz
       y obtenerlug está al principio. */
    if(colocarlug+1==obtenerlug |
       ((colocarlug==q.length-1) & (obtenerlug==0))) {
      System.out.println(" - La cola se ha llenado.");
      return;
    }

    colocarlug++;
    if(colocarlug==q.length) colocarlug = 0; // recorre el bucle hacia atrás
    q[colocarlug] = ch;
  }

  // obtiene un carácter de la cola.
  public char obtener() {
    if(obtenerlug == colocarlug) {
      System.out.println(" - La cola se encuentra vacía.");
      return (char) 0;
    }
obtenerlug++;
    if(obtenerlug==q.length) obtenerlug = 0; // recorre el bucle hacia atrás
```

```
    return q[obtenerlug];
  }
}
```

La cola circular funciona al reciclar el espacio de la matriz que se libera cuando se recuperan los elementos. De esta manera puede almacenar una cantidad ilimitada de elementos, siempre y cuando se vayan eliminado elementos. Aunque es conceptualmente simple (sólo restablece el índice apropiado a cero cuando se alcanza el final de la matriz), las condiciones de límite son un poco confusas al principio. En una cola circular, la cola no queda llena cuando se alcanza el final de la matriz, sino cuando el almacenamiento de un elemento ocasiona que se sobrescriba un elemento no recuperado. Así, **colocar( )** debe verificar varias condiciones para determinar si la cola está llena. Como los comentarios lo sugieren, la cola está llena cuando **colocarlug** es uno menos que **obtenerlug**, o cuando **colocarlug** está al final de la matriz y **obtenerlug** está al principio. Como antes, la cola está vacía cuando **obtenerlug** y **colocarlug** son iguales.

5. Coloque la clase **ColaDin** que se muestra a continuación dentro de **ICDemo.java**. Éste implementa una cola "creciente" que crece cuando el espacio se acaba.

```
// Una cola dinámica.
class ColaDin implements ICCar {
  private char q[]; // esta matriz contiene la cola
  private int colocarlug, obtenerlug; // los índices colocar y obtener

  // Construye una cola vacía dado su tamaño.
  public ColaDin(int dim) {
    q = new char[dim+1]; // asigna memoria a la cola
    colocarlug = obtenerlug = 0;
  }

  // Coloca un carácter en la cola.
  public void colocar(char ch) {
    if(colocarlug==q.length-1) {
      // aumenta el tamaño de la cola
      char t[] = new char[q.length * 2];

      // copia elementos en una nueva cola
      for(int i=0; i < q.length; i++)
        t[i] = q[i];

      q = t;
    }

    colocarlug++;
    q[colocarlug] = ch;
  }

  // obtiene un carácter de la cola.
```

(continúa)

```
  public char obtener() {
    if(obtenerlug == colocarlug) {
      System.out.println(" - La cola se encuentra vacía.");
      return (char) 0;
    }

    obtenerlug++;
    return q[obtenerlug];
  }
}
```

En esta implementación de cola, cuando la cola está llena, un intento de almacenar otro elemento hace que se asigne una nueva matriz que es dos veces más grande que la original, que se copie el contenido actual de la cola en esa matriz y que se almacene una referencia a la nueva matriz en **q**.

6. Para demostrar las tres implementaciones de **ICCar.java**, ingrese la siguiente clase en **ICDemo.java**. Éste usa la referencia a **ICCar.java** para acceder a las tres colas.

```
// Demuestra la interfaz ICCar.
class ICDemo {
  public static void main(String args[]) {
    ColaFija q1 = new ColaFija(10);
    ColaDin q2 = new ColaDin(5);
    ColaCircular q3 = new ColaCircular(10);

    ICCar iC;

    char ch;
    int i;

    iC = q1;
    // Coloca algunos caracteres en una cola fija.
    for(i=0; i < 10; i++)
      iC.colocar((char) ('A' + i));

    // Muestra la cola.
    System.out.print("Contenido de la cola fija: ");
    for(i=0; i < 10; i++) {
      ch = iC.obtener();
      System.out.print(ch);
    }
    System.out.println();

    iC = q2;
    // Coloca algunos caracteres en la cola dinámica.
    for(i=0; i < 10; i++)
      iC.colocar((char) ('Z' - i));
```

```
    // Muestra la cola.
    System.out.print("Contenido de la cola dinámica: ");
    for(i=0; i < 10; i++) {
      ch = iC.obtener();
      System.out.print(ch);
    }

    System.out.println();

    iC = q3;
    // Coloca algunos caracteres en la cola circular.
    for(i=0; i < 10; i++)
      iC.colocar((char) ('A' + i));

    // Muestra la cola.
    System.out.print("Contenido de la cola circular: ");
    for(i=0; i < 10; i++) {
      ch = iC.obtener();
      System.out.print(ch);
    }

    System.out.println();

    // Coloca algunos caracteres en la cola circular.
    for(i=10; i < 20; i++)
      iC.colocar((char) ('A' + i));

    // Muestra la cola.
    System.out.print("Contenido de la cola circular: ");
    for(i=0; i < 10; i++) {
      ch = iC.obtener();
      System.out.print(ch);
    }

    System.out.println("\nAlmacena y consume" +
                       " de una cola circular.");

    // Usa y consume a partir de una cola circular.
    for(i=0; i < 20; i++) {
      iC.colocar((char) ('A' + i));
      ch = iC.obtener();
      System.out.print(ch);
    }

  }
}
```

(continúa)

7. La salida de este programa se muestra a continuación:

```
Contenido de la cola fija: ABCDEFGHIJ
Contenido de la cola dinámica: ZYXWVUTSRQ
Contenido de la cola circular: ABCDEFGHIJ
Contenido de la cola circular: KLMNOPQRST
Almacena y consume a partir de una cola circular.
ABCDEFGHIJKLMNOPQRST
```

8. He aquí algunos aspectos que puede probar por su cuenta: crear una versión circular de **ColaDin**; agregar un método **restablecer()** a **ICCar.java** que restablezca la cola; y crear un método **static** que copie el contenido de un tipo de cola en otro.

HABILIDAD FUNDAMENTAL 8.9

Variables en interfaces

Como ya se mencionó, las variables pueden declararse en una interfaz, pero son implícitamente **public**, **static** y **final**. A primera vista, podría pensarse que existe un uso muy limitado para esas variables, pero es todo lo contrario. Por lo general los programas grandes emplean varios valores de constantes que describen aspectos como el tamaño de la matriz, diversos límites, valores especiales y aspectos parecidos. Debido a que un programa grande suele contener varios archivos fuente separados, es necesario que existan maneras eficientes de lograr que estas constantes estén disponibles para cada archivo. En Java, las variables de interfaz ofrecen una solución.

Para definir un conjunto de constantes compartidas, simplemente cree una **interface** que contenga sólo estas constantes, sin ningún método. Cada archivo que requiera acceder a las constantes simplemente "implementa" la interfaz. Esto trae a la vista las constantes. He aquí un ejemplo simple:

```
// Una interfaz que contiene constantes.
interface IConst {
  int MIN = 0;
  int MAX = 10;
  String MSJERROR = "Error de límites";
}

class IConstD implements IConst {
  public static void main(String args[]) {
    int nums[] = new int[MAX];

    for(int i=MIN; i < 11; i++) {
      if(i >= MAX) System.out.println(MSJERROR);
      else {
        nums[i] = i;
        System.out.print(nums[i] + " ");
      }
    }
  }
}
```

Son constantes.

Pregunte al experto

P: **Cuando convierto un programa de C++ a Java, ¿cómo debo manejarlas instrucciones** #define **en un archivo de encabezado de estilo C++?**

R: La respuesta de Java a los archivos de encabezado y a **#define** que se encuentran en C++ es la interfaz y las variables de interfaz. Para trasladar un archivo de encabezado, simplemente realice una traducción uno a uno.

HABILIDAD FUNDAMENTAL 8.10

Las interfaces pueden extenderse

Una interfaz puede heredar otra con el uso de la palabra clave **extends**. La sintaxis es la misma que para las clases que heredan. Cuando una clase implementa una interfaz que hereda a otra, debe proporcionar implementación para todos los métodos definidos dentro de la cadena de herencia de la interfaz. A continuación se presenta un ejemplo:

```
// Una interfaz puede extender otra.
interface A {
  void met1();
  void met2();
}

// B ahora incluye met1() y met2(): agrega met3().
interface B extends A {
  void met3();
}

// Esta clase debe implementar todo lo de A y B
class MiClase implements B {
  public void met1() {
    System.out.println("Implementa met1().");
  }

  public void met2() {
    System.out.println("Implementa met2().");
  }

  public void met3() {
    System.out.println("Implementa met3().");
```

B hereda a **A**.

```
  }
}

class SiExtend {
  public static void main(String arg[]) {
    MiClase ob = new MiClase();

    ob.met1();
    ob.met2();
    ob.met3();
  }
}
```

Como experimento, podría probar la eliminación de la implementación de **met1()** en **MiClase**. Esto causaría un error en tiempo de compilación. Como ya se comentó, cualquier clase que implemente una interfaz debe implementar todos los métodos definidos por dicha interfaz, incluyendo cualquiera que se herede de otras interfaces.

Aunque los ejemplos que hemos incluido en este libro no emplean con frecuencia los paquetes o las interfaces, ambas herramientas son parte importante del entorno de programación de Java. Casi todos los programas y los applets reales que escriba en Java estarán contenidos dentro de paquetes. Algunos probablemente también implantarán interfaces. Es importante, por lo tanto, que se sienta cómodo con su uso.

Comprobación de dominio del módulo 8

1. Mediante el código del proyecto 8.1, coloque la interfaz **ICCar.java** y sus tres implementaciones en un paquete llamado **PaqueteC**. Manteniendo la demostración de la cola de la clase **ICDemo** en el paquete predeterminado, muestre cómo importar y usar las clases de **PaqueteC**.
2. ¿Qué es un espacio de nombre? ¿Por qué es importante que Java le permita dividir el espacio de nombre?
3. Los paquetes están almacenados en _____.
4. Explique la diferencia entre el acceso **protected** y el predeterminado.
5. Explique las dos maneras en que otros paquetes pueden usar a los miembros de un paquete.
6. “Una interfaz, varios métodos” es un concepto clave de Java. ¿Qué característica lo ejemplifica mejor?
7. ¿Cuántas clases pueden implementar una interfaz? ¿Cuántas interfaces puede implementar una clase?
8. ¿Es posible extender las interfaces?

9. Cree una interfaz para la clase **Automotor** del módulo 7. Llámela **IAutomotor**.
10. Las variables declaradas en una interfaz son implícitamente **static** y **final**. ¿Para qué sirven?
11. En esencia, un paquete es un contenedor de clases. ¿Cierto o falso?
12. ¿Qué paquete estándar de Java se importa automáticamente en un programa?

Módulo 9

Manejo de excepciones

HABILIDADES FUNDAMENTALES

En este módulo se analizará el manejo de excepciones. Una excepción es un error que ocurre en tiempo de ejecución. Con el uso del subsistema de manejo de excepciones usted puede, de una manera estructurada y controlada, manejar errores en tiempo de ejecución. Aunque casi todos los lenguajes modernos de programación ofrecen alguna forma de manejo de excepciones, el soporte de Java es más limpio y flexible que el de casi todos los demás.

Una ventaja importante del manejo de excepciones es que vuelve automático a gran parte del código de manejo de errores que antes se tenía que ingresar "a mano" en cualquier programa largo. Por ejemplo, en algunos lenguajes de cómputo, los códigos de error se regresan cuando un método falla, y estos valores deben revisarse manualmente cada vez que se llama al método. Este método es tedioso y propenso a errores. El manejo de excepciones mejora el manejo de errores al permitir que su programa defina un bloque de código llamado *manejador de excepciones*, que se ejecuta automáticamente cuando ocurre un error. No es necesario comprobar manualmente el éxito o la falla de cada operación o cada llamada específica a un método. Si ocurre un error, éste será procesado por el manejador de excepciones.

Otra razón por la que es importante el manejo de excepciones es que Java define excepciones estándar para errores comunes de programa, como la división entre cero o el archivo no encontrado. Para responder a estos errores, su programa debe vigilar y manejar estas excepciones. Además, la biblioteca de la API de Java hace un uso extenso de las excepciones.

En el análisis final, convertirse en un programador exitoso en Java significa que usted sea completamente capaz de recorrer el subsistema de manejo de excepciones de Java.

HABILIDAD FUNDAMENTAL 9.1

La jerarquía de excepciones

En Java, todas las excepciones están representadas por clases. Todas las clases de excepción se derivan de una clase llamada **Throwable**; así que, cuando una excepción en un programa ocurre, se genera un objeto de algún tipo de clase de excepción. Hay dos subclases directas de **Throwable: Exception** y **Error**. Las excepciones de tipo **Error** están relacionadas con errores que ocurren en la propia máquina virtual de Java, y no en su programa. Estos tipos de excepciones van más allá de su control y por lo general su programa no tratará con ellos. Es por ello que estos tipos de excepciones no se describen aquí.

Los errores que resultan de la actividad del programa están representados por subclases de **Exception**. Por ejemplo, la división entre cero, el límite de matriz y los errores de archivo se encuentran en esta categoría. En general, su programa debe manejar las excepciones de estos tipos. Una subclase importante de **Exception** es **RuntimeException**, la cual se usa para representar varios tipos comunes de errores en tiempo de ejecución.

HABILIDAD FUNDAMENTAL 9.2

Fundamentos del manejo de excepciones

El manejo de excepciones de Java se maneja mediante cinco palabras clave: **try, catch, throw, throws** y **finally**. Forman un subsistema interrelacionado en que el uso de uno implica el uso de otro. En todo

este módulo, se examinarán de manera detallada todas estas palabras clave. Sin embargo, es útil tener desde el principio una comprensión general del papel que juega cada una en el manejo de excepciones. En resumen, he aquí cómo funciona.

Las instrucciones del programa que desee monitorear para excepciones están contenidas dentro del bloque **try**. Si ocurre una excepción dentro del bloque **try**, se *lanza*. Su código puede capturar esta excepción usando **catch** y manejándolo de alguna manera racional. El sistema en tiempo de ejecución de Java lanza automáticamente las excepciones generadas por el sistema. Para lanzar manualmente una excepción, use la palabra clave **throw**. En algunos casos, una excepción que se lanza fuera de un método debe especificarse como tal mediante una cláusula **throws**. Cualquier código que tenga que ejecutarse obligatoriamente una vez que se salga de un bloque **try**, se colocará en un bloque **finally**.

Pregunte al experto

P: **Sólo para estar seguro, ¿podría revisar las condiciones que hacen que una excepción se genere?**

R: Las excepciones se generan de tres maneras. En primer lugar, la máquina virtual de Java puede generar una excepción como respuesta a algún error interno que se encuentre más allá de su control. Por lo general, su programa no manejará estos tipos de excepciones. En segundo lugar, las excepciones estándar, como las que corresponden a la división entre cero o a un índice de matriz fuera de los límites, son generados por errores en el código del programa. Usted necesita manejar estas excepciones. En tercer lugar, puede generar manualmente una excepción empleando la instrucción **throw**. No importa cómo se genere una excepción, ésta se maneja de la misma manera.

Uso de try y catch

En el eje del manejo de excepciones se encuentran **try** y **catch**. Estas palabras clave funcionan juntas: no puede tener una sin la otra. He aquí la forma general de los bloques de manejo de excepciones **try/catch**:

```
try {
  // bloque de código para monitorear errores.
}

catch (TipoExcep1 exOb) {
  // manejador para ExcepTipo1
}
```

```
catch (TipoExcep2 exOb) {
  // manejador para ExcepTipo2
}
.
.
.
```

Aquí, *TipoExcep* es el tipo de excepción que ha ocurrido. Cuando se lanza una excepción, ésta es capturada por la instrucción **catch** correspondiente, que luego procesa la excepción. Como lo muestra la forma general, puede hacer más de una instrucción **catch** asociada con una **try**. El tipo de excepción determina cuál instrucción **catch** se ejecuta. Es decir, si el tipo especificado de excepción por una instrucción **catch** coincide con el de la excepción, entonces esa instrucción **catch** se ejecuta (y se omiten todas las demás). Cuando se captura una excepción, *exOb* recibe su valor.

He aquí un tema importante: si no se lanza una excepción, entonces un bloque **try** terminará de manera normal y se omitirán todas las instrucciones **catch**. Por lo tanto, las instrucciones **catch** sólo se ejecutan si se lanza una excepción.

Un ejemplo simple de excepción

He aquí un ejemplo simple que ilustra la manera de vigilar y capturar una excepción. Como ya lo sabe, es un error tratar de indizar una matriz más allá de sus límites. Cuando esto ocurre, la JVM lanza una **ArrayIndexOutOfBoundsException**. El siguiente programa genera este tipo de excepción a propósito y luego la captura.

```
// Demuestra el manejo de excepciones.
class ExcDemo1 {
  public static void main(String args[]) {
    int nums[] = new int[4];

    try {
      System.out.println("Antes de que se genere la excepción.");

      // Genera una excepción de índice fuera de límites.
      nums[7] = 10;
      System.out.println("Esto no se muestra");
    }
    catch (ArrayIndexOutOfBoundsException exc) {
      // captura la excepción
      System.out.println("¡Índice fuera de límite!");
    }
    System.out.println("Tras la instrucción catch.");
  }
}
```

Intento para indexar límites de **nums** pasados.

Este programa despliega la siguiente salida:

```
Antes de que se genere la excepción.
¡Índice fuera de límite!
Tras la instrucción catch.
```

Aunque es muy corto, el programa anterior ilustra varios elementos clave acerca del manejo de excepciones. En primer lugar, el código que quiere monitorear en busca de errores está contenido dentro de un bloque **try**. En segundo lugar, cuando una excepción ocurre (en este caso, debido al intento del índice **nums** de rebasar sus límites), la excepción se lanza fuera del bloque **try** y es capturada por la instrucción **catch**. En este punto, el control pasa a **catch** y el bloque **try** se termina, es decir, *no* se llama a **catch**. En cambio, se le transfiere la ejecución del programa. Por consiguiente, la instrucción **println()** que sigue al índice fuera de límites nunca se ejecutará. Después de que se ejecuta la instrucción **catch**, el control del programa sigue en las instrucciones posteriores a **catch**. Por lo tanto, su manejador de excepciones tiene el trabajo de remediar el problema causado por la excepción, de modo que la ejecución del programa pueda continuar normalmente.

Recuerde que si el bloque **try** no lanza una excepción, no se ejecutarán instrucciones **catch** y el control del programa se reanudará después de la instrucción **catch**. Para confirmar esto, cambie en el programa anterior la línea

```
nums[7] = 10;
```

por

```
nums[0] = 10;
```

Ahora no se generará una excepción y no se ejecutará el bloque **catch**.

Es importante comprender que todo el código dentro del bloque **try** se monitorea en busca de excepciones. Esto incluye excepciones generadas por un método llamado dentro del bloque **try**. Una excepción lanzada por un método llamado dentro de un bloque **try** puede ser capturada por las instrucciones **catch** asociadas con ese bloque (suponiendo, por supuesto, que el método no captura la propia excepción). Por ejemplo, el siguiente es un programa válido:

```
 /* Una excepción puede ser generada por un
   método y capturada por otro. */

class ExcPrueba {
  // Genera una excepción.
  static void genExc() {
    int nums[] = new int[4];

    System.out.println("Antes de que se genere la excepción.");

    // genera una excepción de índice fuera de límites
```

```
    nums[7] = 10;
    System.out.println("Esto no se muestra");
  }
}

class ExcDemo2 {
  public static void main(String args[]) {

    try {
      ExcPrueba.genExc();
    }
    catch (ArrayIndexOutOfBoundsException exc) {
      // captura la excepción
      System.out.println("¡Índice fuera de límites!");
    }
    System.out.println("Tras la instrucción catch.");
  }
}
```

Excepción generada aquí.

Excepción capturada aquí.

Este programa produce la siguiente salida, que es la misma que la producida por la primera versión del programa que se mostró antes.

```
Antes de que se genere la excepción.
¡Índice fuera de límite!
Tras la instrucción catch.
```

Como **genExc()** es llamado dentro de un bloque **try**, la excepción que se genera (y no se captura) es capturada por la instrucción **catch** de **main()**. Sin embargo, note que si **genExc()** hubiera capturado la propia excepción, nunca hubiera regresado a **main()**.

Comprobación de avance

1. ¿Qué es una excepción?
2. ¿De qué instrucción debe ser parte el código que monitorea en busca de excepciones?
3. ¿Qué hace **catch**? Después de que se ejecuta **catch**, ¿qué le pasa al flujo de la ejecución?

1. Una excepción es un error en tiempo de ejecución.
2. Para que monitoree excepciones debe ser parte del bloque **try**.
3. La instrucción **catch** recibe excepciones. No se llama a una instrucción **catch**; por lo tanto, la ejecución no regresa al punto en que se generó la excepción. En cambio, la ejecución continúa después del bloque **catch**.

HABILIDAD FUNDAMENTAL 9.3

Las consecuencias de una excepción no capturada

Capturar una de las excepciones estándar de Java, como lo hace el programa anterior, conlleva un beneficio secundario: evita la terminación anormal del programa. Cuando se lanza una excepción, alguna pieza del código debe capturarla en alguna parte. En general, si su programa no captura una excepción, entonces ésta será capturada por la JVM. El problema es que el manejador de excepciones predeterminado de la JVM termina la ejecución y despliega una pila de mensajes de rastreo y error. Por ejemplo, en esta versión del ejemplo anterior, la excepción de índice fuera de límite no es capturada por el programa.

```
// Que JVM maneje el error.
class NoManejada {
  public static void main(String args[]) {
    int nums[] = new int[4];

    System.out.println("Antes de que se genere la excepción.");

    // genera una excepción de índice fuera de límite
    nums[7] = 10;
  }
}
```

Cuando el error en el índice de la matriz ocurre, la ejecución se detiene y se despliega el siguiente mensaje de error.

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 7
        at NoManejada.main(NoManejada.java:9)
```

Si bien este mensaje le puede resultar útil mientras depura, no será algo que quiera que los demás vean. Por eso es importante que su programa maneje las excepciones por sí solo, en lugar de depender de la JVM.

Como se mencionó antes, el tipo de excepción debe coincidir con el tipo especificado en la instrucción **catch**. De no ser así, la excepción no se capturará. Por ejemplo, el siguiente programa trata de capturar un error de límite de matriz con una instrucción **catch** para una **ArithmeticException** (otra de las excepciones integradas de Java). Cuando el límite de la matriz se sobrepasa, se genera una **ArrayIndexOutOfBoundsException**, pero ésta no será capturada por la instrucción **catch**. Esto da como resultado una terminación anormal.

```
// ¡Esto no funciona!
class NoCoincideTipoExc {
  public static void main(String args[]) {
```

```
    int nums[] = new int[4];

    try {
      System.out.println("Antes de que se genere la excepción.");

      // genera una excepción de índice fuera de límite
      nums[7] = 10;
      System.out.println("Esto no se despliega");
    }

    /* No puede capturar un error de límite de matriz con una
       ArithmeticException. */
    catch (ArithmeticException exc) {
      // captura la excepción
      System.out.println("¡Índice fuera de límite!");
    }
    System.out.println("Tras la instrucción catch.");
  }
}
```

Esto lanza una **ArrayIndexOutOfBoundsException**.

Esto trata de capturarlo con una **ArithmeticException.**

A continuación se muestra la salida:

```
Antes de que se genere la excepción.
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 7
        at NoCoincideTipoExc.main(NoCoincideTipoExc.java:10)
```

Como la salida lo demuestra, una **catch** para **ArithmeticException** no captura una **ArrayIndexOutOfBoundsException**.

Las excepciones le permiten manejar con elegancia los errores

Uno de los beneficios clave del manejo de excepciones es que le permite a su programa responder a un error y seguir ejecutándose. Considere el siguiente ejemplo que divide los elementos de una matriz entre los elementos de otra. Si ocurre una división entre cero, se genera una **ArithmeticException**. En el programa, esta excepción se maneja al reportar el error y continuar con la ejecución. Por lo tanto, el intento de dividir entre cero no origina un error en tiempo de ejecución abrupto que dé como resultado la terminación del programa. En cambio, si permite que el programa siga la ejecución, manejará con elegancia el error.

```
// Maneja el error con ingenio y continúa.
class ExcDemo3 {
  public static void main(String args[]) {
    int numer[] = { 4, 8, 16, 32, 64, 128 };
    int denom[] = { 2, 0, 4, 4, 0, 8 };
```

```
    for(int i=0; i<numer.length; i++) {
      try {
        System.out.println(numer[i] + " / " +
                           denom[i] + " es " +
                           numer[i]/denom[i]);
      }
      catch (ArithmeticException exc) {
        // captura la excepción
        System.out.println("¡No puede dividir entre cero!");
      }
    }
  }
}
```

Ésta es la salida del programa:

```
4 / 2 es 2
¡No puede dividir entre cero!
16 / 4 es 4
32 / 4 es 8
¡No puede dividir entre cero!
128 / 8 es 16
```

Este ejemplo define otro tema importante: una vez que una excepción se maneja, ésta se elimina del sistema. De manera que, en el programa, cada paso por el bucle entra en el bloque **try** de nuevo y todas las excepciones anteriores se han manejado. Esto le permite que su programa maneje errores repetidos.

Comprobación de avance

1. ¿Qué tan importante es el tipo de excepción en una instrucción **catch**?

2. ¿Qué pasa si una excepción no se captura?

3. Cuando ocurre una excepción, ¿qué debe realizar su programa?

1. El tipo de excepción en un **catch** debe coincidir con el tipo de excepción que desee capturar.
2. Una excepción no capturada conduce a una terminación anormal del programa.
3. Un programa debe manejar la excepción de manera racional e ingeniosa, eliminando la causa de la excepción, de ser posible, y continuando enseguida.

HABILIDAD FUNDAMENTAL 9.4

Uso de varias instrucciones catch

Como se estableció antes, puede asociar más de una instrucción **catch** con una **try**. En realidad, es común llevarlo a cabo de esta manera. Sin embargo, cada **catch** debe capturar un tipo diferente de excepción. Por ejemplo, el programa que se muestra aquí captura errores de límite de matriz y de división entre cero.

```
// Uso de varias instrucciones catch.
class ExcDemo4 {
  public static void main(String args[]) {
    // Aquí, numer es más grande que denom.
    int numer[] = { 4, 8, 16, 32, 64, 128, 256, 512 };
    int denom[] = { 2, 0, 4, 4, 0, 8 };

    for(int i=0; i<numer.length; i++) {
      try {
        System.out.println(numer[i] + " / " +
                           denom[i] + " es " +
                           numer[i]/denom[i]);
      }
      catch (ArithmeticException exc) {
// captura la excepción
        System.out.println("¡No puede dividir entre cero!");
      }
      catch (ArrayIndexOutOfBoundsException exc) {
        // captura la excepción
        System.out.println("No hay elemento coincidente.");
      }
    }
  }
}
```

Varias instrucciones **catch**.

El programa produce la siguiente salida:

```
4 / 2 es 2
¡No puede dividir entre cero!
16 / 4 es 4
32 / 4 es 8
¡No puede dividir entre cero!
128 / 8 es 16
No hay elemento coincidente.
No hay elemento coincidente.
```

Como la salida lo confirma, cada instrucción **catch** responde sólo a su propio tipo de excepción.

En general, las expresiones **catch** se revisan en el orden en el que ocurren en el programa. Sólo se ejecuta una instrucción coincidente. Todos los otros bloques de **catch** se ignoran.

HABILIDAD FUNDAMENTAL 9.5

Captura de excepciones de subclases

Hay un tema importante relacionado con varias instrucciones **catch** que se relacionan con las subclases. Una cláusula **catch** de una superclase coincidirá también con cualquiera de sus subclases. Por ejemplo, como la superclase de todas las excepciones es **Throwable**, para capturar todas las excepciones posibles, capture **Throwable**. Si quiere capturar excepciones de un tipo de superclase y de uno de subclase, coloque la subclase primero en la secuencia de **catch**. Si no lo hace, entonces **catch** de la superclase capturará también todas las clases derivadas. Esta regla se impone por sí misma porque si la superclase se coloca primero, un código ilegible se creará debido a que la cláusula **catch** de la subclase no puede ejecutarse nunca. En Java, el código ilegible constituye un error.

Por ejemplo, considere lo siguiente:

```
// Las subclases deben preceder a las superclases en las instrucciones catch.
class ExcDemo5 {
  public static void main(String args[]) {
    // Aquí, numer es más grande que denom.
    int numer[] = { 4, 8, 16, 32, 64, 128, 256, 512 };
    int denom[] = { 2, 0, 4, 4, 0, 8 };

     for(int i=0; i<numer.length; i++) {
      try {
        System.out.println(numer[i] + " / " +
                           denom[i] + " es " +
                           numer[i]/denom[i]);
      }
      catch (ArrayIndexOutOfBoundsException exc) {   <-- Catch subclass.
        // captura la excepción
        System.out.println("No hay elemento coincidente.");
      }
      catch (Throwable exc) {   <-- Catch superclass.
        System.out.println("Ocurrió alguna excepción.");
      }
    }
  }
}
```

Ésta es la salida de este programa:

```
4 / 2 es 2
Ocurrió alguna excepción.
16 / 4 es 4
32 / 4 es 8
Ocurrió alguna excepción.
128 / 8 es 16
No hay elemento coincidente.
No hay elemento coincidente.
```

En este caso, **catch(Throwable)** captura todas las excepciones, excepto **ArrayIndexOutOfBoundsException**.

El tema de la captura de excepciones de una subclase se vuelve más importante cuando crea excepciones por su cuenta.

Pregunte al experto

P: **¿Por qué razón desearía capturar excepciones de una superclase?**

R: Por supuesto, hay varias razones. He aquí un par de ellas. En primer lugar, si agrega una cláusula **catch** que capture excepciones de tipo **Exception**, entonces habrá agregado efectivamente una cláusula tipo "captura todo" a su manejador de excepciones que trata con todas las excepciones relacionadas con el programa. Este tipo de cláusula resultaría útil en una situación en la que deba evitarse la terminación anormal del programa, sin importar lo que ocurra. En segundo lugar, en algunas situaciones, la misma cláusula puede manejar toda una categoría de excepciones. La captura por parte de la superclase de estas excepciones le permite manejar todo sin código duplicado.

HABILIDAD FUNDAMENTAL 9.6

Es posible anidar bloques try

Es posible anidar un bloque **try** dentro de otro. Una excepción generada dentro del bloque **try** interno que no es capturada por una **catch** asociada con ese **try** se propaga al bloque **try** exterior. Por ejemplo, aquí la **ArrayIndexOutOfBoundsException** no es capturada por la **catch** interna, sino por la externa.

```
// Uso de un bloque try anidado.
class TryAnidado {
  public static void main(String args[]) {
    // Aquí, numer es más grande que denom.
    int numer[] = { 4, 8, 16, 32, 64, 128, 256, 512 };
    int denom[] = { 2, 0, 4, 4, 0, 8 };

    try { // try externo
      for(int i=0; i<numer.length; i++) {
        try { // try anidado
          System.out.println(numer[i] + " / " +
                             denom[i] + " es " +
                             numer[i]/denom[i]);
        }
        catch (ArithmeticException exc) {
          // captura la excepción
```

Bloques **try** anidados.

```
          System.out.println("¡No puede dividir entre cero!");
        }
      }
    }
    catch (ArrayIndexOutOfBoundsException exc) {
      // captura la excepción
      System.out.println("No hay elemento coincidente.");
      System.out.println("Error fatal: programa terminado.");
    }
  }
}
```

A continuación se muestra la salida del programa.

```
4 / 2 es 2
¡No puede dividir entre cero!
16 / 4 es 4
32 / 4 es 8
¡No puede dividir entre cero!
128 / 8 es 16
No hay elemento coincidente.
Error fatal: programa terminado.
```

En este ejemplo, una excepción que pueda manejar el bloque **try** interno (en este caso un error de división entre cero) permite la continuación del programa. Sin embargo, un error de límite de matriz es capturado por el **try** externo, lo que hace que el programa termine.

Aunque ciertamente no es la única razón para que haya instrucciones **try** anidadas, el programa anterior destaca un elemento importante que puede generalizarse. A menudo, los bloques **try** anidados se usan para permitir manejar de distintas maneras las diferentes categorías de errores. Algunos tipos de errores resultan catastróficos y no pueden corregirse. Algunos son menores y pueden manejarse de inmediato. Muchos programadores usan un bloque **try** externo para capturar los errores más graves, permitiendo que los bloques **try** internos manejen los menos serios.

Comprobación de avance

1. ¿Puede usarse un bloque **try** para manejar dos o más tipos diferentes de excepciones?
2. ¿La instrucción **catch** de la excepción de una superclase puede también capturar subclases de las superclases?
3. En bloques **try** anidados, ¿qué le sucede a una excepción que no es capturada por el bloque interno?

1. Sí.
2. Sí.
3. Una excepción no capturada por un bloque **try**/**catch** interno se mueve hacia fuera, es decir, al bloque **try** que lo contiene.

HABILIDAD FUNDAMENTAL 9.7

Lanzamiento de una excepción

En los ejemplos anteriores se han capturado excepciones generadas automáticamente por la JVM. Sin embargo, es posible lanzar manualmente una excepción con el uso de la instrucción **throw**. Aquí se muestra su forma general.

throw *exceptOb;*

Aquí, *exceptOb* debe ser un objeto de una clase de excepción derivada de **Throwable**.

He aquí un ejemplo que ilustra la instrucción **throw** si se lanza manualmente una **ArithmeticException**.

```
// Lanza manualmente una excepción.
class ThrowDemo {
  public static void main(String args[]) {
    try {
      System.out.println("Antes del lanzamiento.");
      throw new ArithmeticException();
    }
    catch (ArithmeticException exc) {
      // captura la excepción
      System.out.println("Excepción capturada.");
    }
    System.out.println("Tras el bloque try/catch.");
  }
}
```

A continuación se muestra la salida de este programa:

```
Antes del lanzamiento.
Excepción capturada.
Tras el bloque try/catch.
```

Observe que **ArithmeticException** se creó empleando **new** en la instrucción **throw**. Recuerde que **throw** lanza un objeto. Por lo tanto, debe crear un objeto para que throw lo lance, es decir, no puede lanzar sólo un tipo.

Relanzamiento de una excepción

Una excepción capturada por una instrucción **catch** puede relanzarse para que la pueda capturar una **catch** externa. La razón más probable para este tipo de relanzamiento es permitir que varios manejadores accedan a la excepción. Por ejemplo, tal vez un manejador de excepciones se encargue de un aspecto de una excepción, mientras que un segundo manejador trate otro aspecto. Recuerde que

Pregunte al experto

P: **¿Por qué razón desearía lanzar manualmente una excepción?**

R: Con mayor frecuencia, las excepciones que lanzará serán instancias de clases de excepción que usted haya creado. Como verá mas adelante en este módulo, la creación de sus propias clases de excepción le permite manejar errores en su código como parte de su estrategia general de manejo de excepciones de su programa.

cuando relanza una excepción, ésta no será recapturada por la misma instrucción **catch**, sino que se propagará a la siguiente instrucción **catch**.

El siguiente programa ilustra el relanzamiento de una excepción.

```
// Relanza una excepción.
class Rethrow {
  public static void genExc() {
    // Aquí, numer es más grande que denom.
    int numer[] = { 4, 8, 16, 32, 64, 128, 256, 512 };
    int denom[] = { 2, 0, 4, 4, 0, 8 };

    for(int i=0; i<numer.length; i++) {
      try {
        System.out.println(numer[i] + " / " +
                           denom[i] + " es " +
                           numer[i]/denom[i]);
      }
      catch (ArithmeticException exc) {
        // captura la excepción
        System.out.println("¡No puede dividir entre cero!");
      }
      catch (ArrayIndexOutOfBoundsException exc) {
        // captura la excepción
        System.out.println("No hay elemento coincidente.");
        throw exc; // relanza la excepción
      }
    }
  }
}

class RethrowDemo {
  public static void main(String args[]) {
    try {
      Rethrow.genExc();
```

Relanza la excepción.

```
    }
    catch(ArrayIndexOutOfBoundsException exc) {
      // recaptura la excepción
     System.out.println("Error fatal: " +
                        "programa terminado.");
    }
  }
}
```

Captura la excepción lanzada.

En este programa, los errores de división entre cero se manejan localmente mediante **getExc()**; sin embargo, un error de límite de matriz se relanza. En este caso, es capturado por **main()**.

Comprobación de avance

1. ¿Qué función realiza **throw**?
2. ¿**throw** lanza tipos u objetos?
3. ¿Es posible relanzar una excepción después de que ésta se ha capturado?

HABILIDAD FUNDAMENTAL 9.8

Análisis detallado de Throwable

Hasta este punto, hemos estado capturando excepciones, pero no hemos realizado ninguna acción respecto al propio objeto de la excepción. Como se mostró en todos los ejemplos anteriores, una cláusula **catch** especifica un tipo de excepción y un parámetro. El parámetro recibe el objeto de la excepción. Debido a que todas las excepciones son subclases de **Throwable**, todas las excepciones soportan los métodos definidos por **Throwable**. En la tabla 9.1 se muestran algunos de uso más común.

De los métodos definidos por **Throwable**, los tres de mayor interés son **printStackTrace()**, **getMessage()** y **toString()**. Si llama a **printStackTrace()**, podrá desplegar el mensaje de error estándar más un registro de las llamadas a método que llevan a la excepción. Para obtener el mensaje de error estándar de una excepción, llame a **getMessage()**. Como opción, puede usar **toString()** para recuperar el mensaje estándar. También se llama al método **toString()**cuando se usa una excepción como archivo para **println()**. El siguiente programa demuestra estos métodos.

1. **throw** genera una excepción.
2. **throw** lanza objetos. Por supuesto, estos objetos deben ser instancias de clases de excepciones válidas.
3. Sí.

Método	Descripción
Throwable fillInStackTrace()	Regresa un objeto de **Throwable** que contiene un rastreo de pila completo. Es posible relanzar este objeto.
String getLocalizedMessage()	Regresa una descripción localizada de la excepción.
String getMessage()	Regresa una descripción de la excepción.
void printStackTrace()	Despliega el rastreo de la pila.
void printStackTrace(PrintStream *flujo*)	Envía el rastreo de la pila al flujo especificado.
void printStackTrace(PrintWriter *flujo*)	Envía el rastreo de la pila al flujo especificado.
String toString()	Regresa un objeto **String** que contiene una descripción de la excepción. Este método es llamado por **println( )** cuando da salida a un objeto de **Throwable**.

Tabla 9.1 Métodos de uso común definidos por **Throwable**.

```
// Uso de métodos de Throwable.

class ExcPrueba {
  static void genExc() {
    int nums[] = new int[4];

    System.out.println("Antes de que se genere la excepción.");

    // genera una excepción de índice fuera de límite
    nums[7] = 10;
    System.out.println("Esto no se despliega");
  }
}

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

    try {
      ExcPrueba.genExc();
    }
    catch (ArrayIndexOutOfBoundsException exc) {
      // captura la excepción
      System.out.println("El mensaje estándar es: ");
      System.out.println(exc);
      System.out.println("\nRastreo de pila: ");
      exc.printStackTrace(); //"¡Índice fuera de límite!");
    }
```

```
    System.out.println("Tras la instrucción catch.");
  }
}
```

Ésta es la salida de este programa:

```
Antes de que se genere la excepción.
El mensaje estándar es:
java.lang.ArrayIndexOutOfBoundsException: 7

Rastreo de pila:
java.lang.ArrayIndexOutOfBoundsException: 7
        at ExcPrueba.genExc(UsoMétodosThrowable.java:10)
        at UsoMétodosThrowable.main(UsoMétodosThrowable.java:19)
Tras la instrucción catch.
```

HABILIDAD FUNDAMENTAL 9.9

Uso de finally

Es posible que en ocasiones desee definir un bloque de código que se ejecute cuando se deje un bloque **try/catch**. Por ejemplo, tal vez una excepción genere un error que termine el método actual causando su regreso prematuro. Sin embargo, ese método puede tener abierto un archivo o una conexión de red que necesite cerrarse. Estos tipos de circunstancias son comunes en programación, pero Java proporciona una manera conveniente de manejarlos: **finally**.

Para especificar que se ejecute un bloque de código cuando se deja un bloque **try/catch**, debe incluir un bloque de **finally** al final de la secuencia **try/catch**. A continuación se muestra la forma generada de una **try/catch** que incluye **finally**.

```
try {
  // bloque de código para monitorear errores.
}

catch (TipoExcep1 exOb) {
  // manejador para ExcepTipo1
}
catch (TipoExcep2 exOb) {
  // manejador para ExcepTipo2
}

// ...
finally {
  // código de finally
}
```

El bloque de **finally** se ejecutará cada vez que la ejecución deje el bloque **try/catch** sin importar la condición que la origine. Es decir, si el bloque **try** termina de manera normal, o debido a una excepción, el último código ejecutado estará definido por **finally**. El bloque de **finally** también se ejecuta si cualquier código dentro del bloque de **try** o cualquiera de las instrucciones de **catch** regresan a partir del método.

He aquí un ejemplo de **finally**.

```
// Uso de finally.
class UsoFinally {
  public static void genExc(int que) {
    int t;
    int nums[] = new int[2];

    System.out.println("Recibiendo " + que);
    try {
      switch(que) {
        case 0:
          t = 10 / que; // genera un error de división entre cero
          break;
        case 1:
          nums[4] = 4; // genera un error de índice de matriz.
          break;
        case 2:
          return; // regresa del bloque try
      }
    }
    catch (ArithmeticException exc) {
      // captura la excepción
      System.out.println("¡No puede dividir entre cero!");
      return; // return from catch
    }
    catch (ArrayIndexOutOfBoundsException exc) {
      // captura la excepción
      System.out.println("No hay elemento coincidente.");
    }
    finally {
      System.out.println("Dejando try.");
    }
  }
}

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

    for(int i=0; i < 3; i++) {
      UsoFinally.genExc(i);
      System.out.println();
```

Esto se ejecuta cuando se sale de los bloques de **try/catch**.

```
    }
  }
}
```

Ésta es la salida de este programa:

```
Recibiendo 0
¡No puede dividir entre cero!
Dejando try.

Recibiendo 1
No hay elemento coincidente.
Dejando try.

Recibiendo 2
Dejando try.
```

Como la salida lo muestra, no importa cómo se salga del bloque de **try**, el bloque de **finally** se ejecutará.

Comprobación de avance

1. ¿Las clases de excepción son subclases de qué tipo de clase?
2. ¿Cuándo se ejecuta el código dentro de un bloque de **finally**?
3. ¿De qué manera puede desplegar un rastreo de pila de los eventos que llevan a una excepción?

HABILIDAD FUNDAMENTAL 9.10

Uso de throws

En algunos casos, si un método genera una excepción que no se maneja, debe declarar esa excepción en una cláusula **throws**. He aquí la forma general de un método que incluye una cláusula **throws**:

tipo-ret nombreMet(*lista-pams*) throws *lista-excepción* {
  // cuerpo
}

Aquí, *lista-excepción* es una lista de excepciones separadas por comas que el método podría lanzar fuera de sí mismo.

Tal vez se pregunte por qué no necesita especificar una cláusula **throws** para alguno de los ejemplos anteriores que lanzaron excepciones fuera de los métodos. La respuesta es que las

1. **Throwable**.
2. Un bloque de **finally** es lo último que se ejecuta cuando se sale de un bloque de **try**.
3. Para imprimir un rastreo de pila, llame a **printStackTrace()**, el cual está definido por **Throwable**.

excepciones que son subclases de **Error** o **RuntimeException** no necesitan especificarse en una lista de **throws**. Java simplemente supone que un método puede lanzar una. Todos los demás tipos de excepciones *sí* necesitan declararse. Si deja de hacerlo se genera un error en tiempo de compilación.

En realidad, lo que usted vio al principio de este bloque fue un ejemplo de una cláusula **throws**. Como recordará, cuando realizó entrada del teclado, necesitó agregar la cláusula

```
throws java.io.IOException
```

a **main()**. A continuación comprenderá por qué: una instrucción de entrada podía generar una **IOException**, por lo que en ese momento no era posible manejar esa excepción. Por lo tanto, esa excepción debía lanzarse fuera de **main()**, así que era necesario especificarlo de esa manera. Ahora que ya sabe más acerca de las excepciones, puede manejar fácilmente una **IOException**.

Veamos un ejemplo que maneja **IOException**, el cual crea un método llamado **aviso()** que despliega un mensaje de aviso y luego lee un carácter del teclado. Debido a que se está llevando a cabo una entrada, una **IOException** podría ocurrir. Sin embargo, el método **aviso()** no maneja una **IOException** por sí mismo, sino que emplea la cláusula **throws**, lo que significa que el método al que llama debe manejarlo. En este ejemplo, el método al que llama es **main()**, el cual se encarga del error.

```
// Uso de throws.
class ThrowsDemo {
  public static char aviso(String cad)
    throws java.io.IOException {

    System.out.print(cad + ": ");
    return (char) System.in.read();
  }

  public static void main(String args[]) {
    char ch;

    try {
      ch = aviso("Escriba una letra");
    }
    catch(java.io.IOException exc) {
      System.out.println("Ocurrió una excepción de E/S.");
      ch = 'X';
    }

    System.out.println("Usted oprimió " + ch);
  }
}
```

Observe la cláusula **throws**.

Como **aviso()** podría lanzar una excepción, debe incluirse una llamada a él dentro de un bloque de **try**.

En un comentario relacionado, observe que **IOException** está plenamente calificado por su nombre de paquete de **java.io**. Como aprenderá en el módulo 10, el sistema de E/S de Java está contenido en el paquete **java.io**. Por lo tanto, la **IOException** también está contenida allí. También hubiera sido posible importar **java.io** y luego hacer referencia a IOException de manera directa.

HABILIDAD FUNDAMENTAL 9.11

Las excepciones integradas de Java

Dentro del paquete estándar **java.lang**, Java define varias clases de excepciones. En este sentido, se han usado unas cuantas en los ejemplos anteriores. Las excepciones más generales son las subclases del tipo estándar **RuntimeException**. Como **java.lang** se importa implícitamente en todos los programas de Java, la mayor parte de las excepciones derivadas de **RuntimeException** están disponibles de manera automática. Más aún, no necesitan ser incluidas en la lista de **throws** de ningún método. En el lenguaje de Java, se les llama *excepciones no revisadas* porque el compilador no revisa si un método maneja o lanza estas excepciones. Las excepciones no revisadas y que son definidas en **java.lang** aparecen en la tabla 9.2. En la tabla 9.3 se presentan las excepciones definidas por **java.lang** que deben incluirse en la lista de **throws** si ese método es capaz de generar una de estas excepciones y no manejarla por sí mismo. A éstas se les denomina *excepciones revisadas*. Java define otros tipos de excepciones que se relacionan con sus diversas bibliotecas de clases, como la **IOException** que ya se mencionó.

Excepción	Significado
ArithmeticException	Error aritmético, como la división entre cero.
ArrayIndexOutOfBoundsException	El índice de la matriz está fuera del límite.
ArrayStoreException	Asignación a un elemento de una matriz de tipo incompatible.
ClassCastException	Moldeo de clase no válido.
IllegalArgumentException	Argumento ilegal usado para invocar un método.
IllegalMonitorStateException	Operación ilegal de monitoreo, como esperar en un subproceso no bloqueado.
IllegalStateException	El entorno o la aplicación está en un estado incorrecto.
IllegalThreadStateException	La operación solicitada no es compatible con el estado del subproceso actual.
IndexOutOfBoundsException	Algún tipo de índice está fuera de los límites.
NegativeArraySizeException	Matriz creada con un tamaño negativo.
NullPointerException	Uso no válido de una referencia a null.
NumberFormatException	Conversión no válida de una cadena a un formato numérico.
SecurityException	Intento de violar la seguridad.
StringIndexOutOfBounds	Intento de indizar fuera de los límites de una cadena.
TypeNotPresentException	Tipo no encontrado (añadido en J2SE 5).
UnsupportedOperationException	Se encontró una operación no soportada.

Tabla 9.2 Las excepciones no revisadas que están definidas en **java.lang**.

Excepción	Significado
ClassNotFoundException	No se encuentra la clase.
CloneNotSupportedException	Intento de clonar un objeto que no implementa la interfaz **Cloneable**.
IllegalAccessException	Se niega el acceso a una clase.
InstantiationException	Intento de crear un objeto de una clase o una interfaz abstracta.
InterruptedException	Un subproceso ha interrumpido otro.
NoSuchFieldException	No existe un campo solicitado.
NoSuchMethodException	No existe un método solicitado.

Tabla 9.3 Las excepciones revisadas que están definidas en **java.lang**.

Pregunte al experto

P: **He oído que Java soporta las llamadas *excepciones encadenadas*. ¿Qué son éstas?**

R: Las excepciones encadenadas son una adición relativamente reciente a Java, pues se incluyeron en 2002 en el J2SE 1.4. La función de las excepciones encadenadas le permite especificar una excepción como la causa de otra. Por ejemplo, imagine una situación en la que un método lanza una **ArithmeticException** debido a un intento de dividir entre cero. Sin embargo, la causa real del problema es que ocurrió un error de E/S, que hizo que el divisor tomara un valor impropio. Aunque el método debe lanzar una **ArithmeticException**, porque ese fue el error que ocurrió, tal vez quiera dejar que el código que llama sepa que la causa fue el error de E/S. Las excepciones encadenadas le permiten manejar ésta y cualquier otra situación en la que existan capas de excepciones.

Para permitir las excepciones encadenadas, se añadieron a **Throwable** dos constructores y dos métodos. A continuación se muestran los constructores:

Throwable(Throwable *causaExc*)

Throwable(String *msj*, Throwable *causaExc*)

En la primera forma, *causaExc* es la excepción que causa la excepción real, es decir, es la razón de que una excepción haya ocurrido. La segunda forma le permite especificar una

(continúa)

descripción al mismo tiempo que especifica una excepción que sea la causa. De igual manera, se han agregado estos dos constructores a las clases **Error**, **Exception** y **RuntimeException**.

Los métodos de excepciones encadenadas que se agregaron a **Throwable** son **getCause( )** e **initCause( )**. A continuación se muestran:

Throwable getCause()

Throwable initCause(Throwable *causaExc*)

El método **getCause( )** regresa la excepción que origina la excepción actual. Si no hay una excepción causante, se regresa **null**. El método **initCause( )** asocia *causaExc* con la excepción que invoca y regresa una referencia a la excepción. Por consiguiente, puede asociar una causa con una excepción después de que se haya creado la excepción. En general, **initCause( )** se utiliza para establecer una causa para clases heredadas de excepción que no soportan los dos constructores adicionales antes descritos. Al momento de escribir esto, muchas excepciones integradas de Java, como **ArithmeticException**, no definen constructores adicionales relacionados con la causa. De ahí que necesite usar **initCause( )** para agregar una cadena de excepciones a dichas excepciones.

Las excepciones encadenadas no son completamente necesarias en todos los programas. Sin embargo, en los casos en los que resulta útil saber la causa, éstas ofrecen una solución ingeniosa.

Comprobación de avance

1. ¿Para qué se utiliza **throws**?
2. ¿Cuál es la diferencia entre las excepciones revisadas y las no revisadas?
3. Si un método genera una excepción que maneje, ¿debe incluir una cláusula **throws** para la excepción?

HABILIDAD FUNDAMENTAL 9.12

Creación de subclases de excepciones

Aunque las excepciones integradas de Java manejan los errores más comunes, el mecanismo de manejo de excepciones de Java no está limitado a estos errores. En realidad, parte de la importancia del método de Java para las excepciones es su capacidad de manejar excepciones creadas por usted y que corresponden a errores en su propio código. La creación de una excepción es fácil: sólo debe

1. Cuando un método genere una excepción que no maneje, deberá establecer el hecho utilizando una cláusula **throws**.
2. No se requiere una cláusula **throws** para excepciones no revisadas.
3. No. Sólo se requiere una cláusula **throws** cuando el método no maneja la excepción.

definir una subclase de **Exception** (que sea, por supuesto, una subclase de **Throwable**). En realidad no es necesario que sus subclases implementen algo: es su existencia en el sistema de tipos lo que le permite usarlas como excepciones.

La clase **Exception** no define métodos por sí misma. Por supuesto, hereda los métodos proporcionados por **Throwable**. Así, todas las excepciones, incluyendo las que usted cree, tienen a su disposición los métodos definidos por **Throwable**. Claro está que puede sobrescribir uno o más de estos métodos en las subclases de excepciones que usted cree.

He aquí un ejemplo que crea una excepción llamada **NotIntResultException**, la cual se genera cuando el resultado de dividir dos valores enteros produce un resultado con un valor fraccionario. **NotIntResultException** tiene dos campos que contienen los valores enteros: un constructor y una sobrescritura del método **toString()**, lo que permite que la descripción de la excepción se despliegue usando **println()**.

```
// Uso de una excepción personalizada.

// Crea una excepción.
class NonIntResultException extends Exception {
  int n;
  int d;

  NonIntResultException(int i, int j) {
    n = i;
    d = j;
  }

  public String toString() {
    return "El resultado de " + n + " / " + d +
           " no es un entero.";
  }
}

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

    // Aquí, numer contiene algunos valores nones.
    int numer[] = { 4, 8, 15, 32, 64, 127, 256, 512 };
    int denom[] = { 2, 0, 4, 4, 0, 8 };
    for(int i=0; i<numer.length; i++) {
      try {
        if((numer[i]%2) != 0)
          throw new
            NonIntResultException(numer[i], denom[i]);

        System.out.println(numer[i] + " / " +
                           denom[i] + " es " +
                           numer[i]/denom[i]);
```

```
      }
      catch (ArithmeticException exc) {
        // captura la excepción
        System.out.println("¡No puede dividir entre cero!");
      }
      catch (ArrayIndexOutOfBoundsException exc) {
        // captura la excepción
        System.out.println("No hay elemento coincidente.");
      }
      catch (NonIntResultException exc) {
        System.out.println(exc);
      }
    }
  }
}
```

Ésta es la salida de este programa:

```
4 / 2 es 2
¡No puede dividir entre cero!
El resultado de 15 / 4 no es un entero.
32 / 4 es 8
¡No puede dividir entre cero!
El resultado de 127 / 8 no es un entero.
No hay elemento coincidente.
No hay elemento coincidente.
```

Pregunte al experto

P: **¿Cuándo debo usar el manejo de excepciones en un programa?; ¿cuándo debo crear mis propias clases de excepciones personalizadas?**

R: Como la API de Java hace un uso extenso de las excepciones para reportar errores, casi todos los programas reales emplean el manejo de excepciones. Ésta es la parte del manejo de excepciones que la mayoría de los nuevos programadores de Java encuentran fácil de usar. Es más difícil decidir cuándo y cómo usar sus propias excepciones personalizadas. En general, los errores pueden reportarse de dos maneras: valores de regreso y excepciones. ¿Cuándo es mejor un método que otro? Para decirlo en pocas palabras, en Java el manejo de excepciones debe ser la norma. Es cierto que el regreso de un código de error constituye una opción válida en algunos casos, pero las excepciones proporcionan una manera más poderosa y estructurada de manejar los errores, ya que son la manera profesional en la que los programadores de Java manejan los errores en su código.

Proyecto 9.1 Adición de excepciones a la clase Cola

CExcDemo

En este proyecto creará dos clases de excepciones que puedan utilizarse en las clases de cola que se desarrollaron en el proyecto 8.1. Estas excepciones indicarán las condiciones de error de cola llena y cola vacía. Además, pueden lanzarse con los métodos **colocar()** y **obtener()**, respectivamente. Por razones de espacio, este proyecto agregará estas excepciones a la clase **ColaFija**, pero puede incorporarlas fácilmente a las otras clases de cola del proyecto 8.1.

Paso a paso

1. Cree un archivo llamado **CExcDemo.java**.
2. En **CExcDemo.java**, defina las siguientes excepciones.

```
/*
    Proyecto 9.1

    Agrega manejo de excepciones a las clases de cola.
*/

// Una excepción para los errores de cola llena.
class ExcepciónColaLlena extends Exception {
  int dim;

  ExcepciónColaLlena(int s) { dim = s; }

  public String toString() {
   return "\nLa cola se ha llenado. El tamaño máximo es " +
          dim;
  }
}

// Una excepción para los errores de cola vacía.
class ExcepciónColaVacía extends Exception {

  public String toString() {
   return "\nLa cola está vacía.";
  }
}
```

Se genera una **ExcepciónColaLlena** cuando se hace el intento de almacenar un elemento en una cola que está llena. Se genera una **ExcepciónColaVacía** cuando se hace el intento de eliminar un elemento de una cola vacía.

(continúa)

3. Modifique la clase **ColaFija** para que lance excepciones cuando un error ocurra, tal y como se muestra aquí. Añádala a **CExcDemo.java**.

```
// Una clase de cola de tamaño fijo para caracteres que utiliza excepciones.
class ColaFija implements ICCar {
  private char q[]; // esta matriz contiene la cola
  private int colocarlug, obtenerlug; // los índices colocar y obtener

  // Construye una cola vacía dado su tamaño.
  public ColaFija(int dim) {
    q = new char[dim+1]; // asigna memoria para la cola
    colocarlug = obtenerlug = 0;
  }

  // Coloca un carácter en la cola.
  public void colocar(char ch)
    throws ExcepciónColaLlena {

    if(colocarlug==q.length-1)
      throw new ExcepciónColaLlena(q.length-1);

    colocarlug++;
    q[colocarlug] = ch;
  }

  // obtiene un carácter de la cola.
  public char obtener()
    throws ExcepciónColaVacía {

    if(obtenerlug == colocarlug)
      throw new ExcepciónColaVacía();

    obtenerlug++;
    return q[obtenerlug];
  }
}
```

Observe que se requieren dos pasos para agregar excepciones a **ColaFija**. Primero **obtener( )** y **colocar( )** deben tener una cláusula **throws** agregada a sus declaraciones. En segundo lugar, cuando un error ocurre, estos métodos lanzan una excepción. El empleo de excepciones permite que el código que llama maneje el error de manera racional. Tal vez recuerde que las versiones anteriores simplemente reportaban el error. Lanzar una excepción es un método mucho mejor.

4. Para probar la clase **ColaFija** actualizada, agregue la clase **CExcDemo** que se muestra aquí a **CExcDemo.java**.

```
// Demuestra las excepciones de la cola.
class CExcDemo {
  public static void main(String args[]) {
    ColaFija q = new ColaFija(10);
    char ch;
    int i;

    try {
      // sobrepasa la cola
      for(i=0; i < 11; i++) {
        System.out.print("Intento de almacenar : " +
                         (char) ('A' + i));
        q.colocar((char) ('A' + i));
        System.out.println("- OK");
      }
      System.out.println();
    }
    catch (ExcepciónColaLlena exc) {
      System.out.println(exc);
    }
    System.out.println();

    try {
      // intento de quitar un elemento de una cola vacía
      for(i=0; i < 11; i++) {
        System.out.print("Obteniendo el siguiente carácter: ");
        ch = q.obtener();
        System.out.println(ch);
      }
    }
    catch (ExcepciónColaVacía exc) {
      System.out.println(exc);
    }
  }
}
```

5. Debido a que **ColaFija** implementa la interfaz **ICCar**, que define los dos métodos de la cola **obtener()** y **colocar()**, se necesitará cambiar **ICCar** para reflejar la clásula **throws**. He aquí la interfaz **ICCar** actualizada. Recuerde que esto debe estar en un archivo independiente llamado **ICCar.java**.

```
// Una interfaz de cola de caracteres que lanza excepciones.
public interface ICCar {
  // Coloca un carácter en la cola.
  void colocar(char ch) throws ExcepciónColaLlena;
```

(continúa)

```
  // Obtiene un carácter de la cola.
  char obtener() throws ExcepciónColaVacía;
}
```

6. Ahora compile el archivo actualizado **ICCar.java**. A continuación, compile **CExcDemo.java**. Por último, ejecute **CExcDemo**. Verá la siguiente salida:

```
Intento de almacenar : A: OK
Intento de almacenar : B: OK
Intento de almacenar : C: OK
Intento de almacenar : D: OK
Intento de almacenar : E: OK
Intento de almacenar : F: OK
Intento de almacenar : G: OK
Intento de almacenar : H: OK
Intento de almacenar : I: OK
Intento de almacenar : J: OK
Intento de almacenar : K
La cola se ha llenado. El tamaño máximo es 10

Obteniendo el siguiente carácter: A
Obteniendo el siguiente carácter: B
Obteniendo el siguiente carácter: C
Obteniendo el siguiente carácter: D
Obteniendo el siguiente carácter: E
Obteniendo el siguiente carácter: F
Obteniendo el siguiente carácter: G
Obteniendo el siguiente carácter: H
Obteniendo el siguiente carácter: I
Obteniendo el siguiente carácter: J
Obteniendo el siguiente carácter:
La cola está vacía.
```

Comprobación de dominio del módulo 9

1. ¿Cuál es la clase superior en la jerarquía de excepciones?

2. Explique brevemente cómo usar **try** y **catch**.

3. ¿Qué está incorrecto en este fragmento?

```
// ...
vals[18] = 10;
catch (ArrayIndexOutOfBoundsException exc) {
  // maneja error
}
```

4. ¿Qué pasa si no se captura una excepción?

5. ¿Qué está incorrecto en este fragmento?

```
class A extends Exception { ...

class B extends A { ...

// ...

try {
  // ...
}
catch (A exc) { ... }
catch (B exc) { ... }
```

6. ¿Una excepción capturada por un bloque interno de **catch** puede relanzar dicha excepción a un bloque de **catch** externo? Explique su respuesta.

7. El bloque de **finally** es el último fragmento de código ejecutado antes de que su programa termine. ¿Cierto o falso?

8. ¿Qué tipo de excepción debe declararse explícitamente en una cláusula **throw** de un método?

9. ¿Qué está incorrecto en este fragmento?

```
class MiClase { // ... }
// ...
throw new MiClase();
```

10. En la pregunta 3 de la Comprobación de dominio del módulo 6 creó una clase **Pila**. Agregue excepciones personalizadas a su clase que reporten las condiciones de pila llena y pila vacía.

11. ¿Cuáles son las tres maneras en las que una excepción puede generarse?

12. ¿Cuáles son las dos subclases directas de **Throwable**?

Módulo 10

Uso de E/S

HABILIDADES FUNDAMENTALES

10.1	Comprenda lo que es un flujo
10.2	Conozca la diferencia entre los flujos de bytes y caracteres
10.3	Conozca las clases de flujos de bytes de Java
10.4	Conozca las clases de flujos de caracteres de Java
10.5	Conozca los flujos predeterminados
10.6	Use flujos de bytes
10.7	Use flujos de bytes para E/S de archivos
10.8	Lea y escriba datos binarios
10.9	Use archivos de acceso directo
10.10	Use flujos de caracteres
10.11	Use flujos de caracteres para E/S de archivo
10.12	Aplique los envolventes de tipo de Java para convertir cadenas numéricas

Desde el principio del libro ha estado usando partes del sistema de E/S de Java como **println()**. Sin embargo, ha procedido sin mayor explicación formal. Debido a que el sistema de E/S de Java está basado en una jerarquía de clases, no era posible presentar la teoría sin analizar primero las clases, la herencia y las excepciones. Ahora es el momento de examinar el método de E/S de Java con detenimiento.

Debe estar consciente de que el sistema de E/S de Java es muy grande y que contiene muchas clases, interfaces y métodos. Este tamaño se debe a que Java define dos sistemas completos de E/S: uno para la E/S de bytes y otro para la de caracteres. No será posible analizar aquí cada aspecto de la E/S de Java (¡fácilmente podría dedicarse un libro completo al sistema de E/S de Java!). Sin embargo, en este módulo se proporcionará una introducción a las funciones más importantes y de uso más común. Por fortuna, el sistema de E/S de Java es coherente y consistente; una vez que comprenda sus fundamentos, podrá dominar con facilidad el resto del sistema de E/S.

En este módulo se examinará el método de Java para la E/S de consola y de archivo. Sin embargo, antes de que iniciar, es importante destacar un tema del que se habló en páginas anteriores: casi ninguna de las aplicaciones reales de Java es un programa de texto y consola, sino programas orientados gráficamente, como applets, que dependen de una interfaz tipo Windows para la interacción con el usuario. Por lo tanto, la parte del sistema de E/S de Java que se relaciona con la entrada y la salida en la consola no se utiliza ampliamente en el código comercial. Aunque los programas que están basados en texto son excelentes como ejemplos de enseñanza, no tienen un uso real o importante en Java. En el módulo 14 verá cómo se crean los applets y aprenderá los fundamentos de soporte de Java para las interfaces gráficas de usuario.

HABILIDAD FUNDAMENTAL 10.1

La E/S de Java está construida sobre flujos

Los programas de Java realizan su E/S a través de flujos. Un *flujo* es una abstracción que produce o consume información. Un flujo está vinculado con un dispositivo físico mediante el sistema de E/S de Java. Todos los flujos se comportan de la misma manera, aunque los dispositivos físicos a los que están conectados sean distintos. De modo que los mismos métodos y clases de E/S se pueden aplicar a cualquier tipo de dispositivo. Por ejemplo, los mismos métodos que usa para escribir en la consola pueden emplearse también para escribir en un archivo de disco. Java implementa los flujos dentro de jerarquías definidas en el paquete **java.in.**

HABILIDAD FUNDAMENTAL 10.2

Flujos de bytes y de caracteres

Las versiones modernas de Java definen dos tipos de flujos: de bytes y de caracteres (la versión original de Java sólo definía el flujo de bytes; sin embargo, los de caracteres se agregaron rápidamente). Los flujos de bytes proporcionan una manera eficiente de manejar la entrada y salida de bytes. Por ejemplo, se usan cuando se leen o escriben datos binarios y son especialmente útiles cuando se trabaja con archivos. Los flujos de caracteres están diseñados para manejar la entrada y salida de caracteres. Usan Unicode y, por consiguiente, pueden tener un uso internacional. Además, en algunos casos, los flujos de caracteres resultan más eficientes que los de bytes.

El hecho de que Java defina dos tipos diferentes de flujos hace que el sistema de E/S sea muy grande porque se requieren dos conjuntos separados de jerarquías de clases (una para bytes y la otra para caracteres). El simple número de clases de E/S puede hacer que el sistema de E/S parezca más intimidante de lo que realmente es. Sólo recuerde que, en su mayor parte, la funcionalidad de los flujos de bytes actúa de manera paralela a la de los flujos de caracteres.

Un tema adicional: en su nivel más bajo, toda la E/S está orientada a bytes. Los flujos de caracteres simplemente proporcionan un medio conveniente y eficiente para manejar caracteres.

HABILIDAD FUNDAMENTAL

10.3 Las clases de flujos de bytes

Los flujos de bytes están definidos por el uso de dos jerarquías de clases. En la parte superior están dos clases abstractas: **InputStream** y **OutputStream**. La primera define las características comunes a los flujos de entrada de bytes, mientras que **OutputStream** describe el comportamiento de los flujos de salida de bytes.

A partir de **InputStream** y **OutputStream** se crean varias subclases concretas que ofrecen una funcionalidad variable y manejan los detalles de lectura y escritura a varios dispositivos, como los archivos de disco. Las clases de flujo de bytes se muestran en la tabla 10.1. No se sienta abrumado por el número de clases diferentes pues una vez que pueda usar un flujo de bytes, le será fácil dominar las demás.

HABILIDAD FUNDAMENTAL

10.4 Las clases de flujo de caracteres

Los flujos de caracteres están definidos por dos jerarquías de clases encabezadas por dos clases abstractas: **Reader** y **Writer**. **Reader** se usa para la entrada y **Writer** para la salida. Las clases concretas derivadas de **Reader** y **Writer** operan sobre flujos de caracteres de Unicode.

A partir de **Reader** y **Writer** se derivan varias subclases concretas que manejan varias situaciones de E/S. En general, las clases de caracteres son paralelas a las de bytes. Las clases de flujo de caracteres se muestran en la tabla 10.2.

HABILIDAD FUNDAMENTAL

10.5 Los flujos predefinidos

Como ya lo sabe, todos los programas de Java importan automáticamente el paquete **java.lang**. Este paquete define una clase llamada **System**, que encapsula varios aspectos del entorno en tiempo de ejecución. Entre otras cosas, contiene tres variables predefinidas de flujo llamadas **in**, **out** y **err**. Estos campos están declarados como **public** y **static** dentro de **System**. Esto significa que otra parte de su programa los puede emplear sin referencia a un objeto específico de **System**.

System.out hace referencia al flujo de salida estándar, la cual, como opción predeterminada, es la consola. **System.in** hace referencia a la entrada estándar, la cual, como opción predeterminada, es el

Clase de flujo de byte	Significado
BufferedInputStream	Flujo de entrada almacenado en el búfer
BufferedOutputStream	Flujo de salida almacenado en el búfer
ByteArrayInputStream	Flujo de entrada que lee a partir de una matriz de bytes
ByteArrayOutputStream	Flujo de salida que escribe a una matriz de bytes
DataInputStream	Flujo de entrada que contiene métodos para leer los tipos estándar de datos de Java
DataOutputStream	Flujo de salida que contiene métodos para escribir los tipos estándar de datos de Java
FileInputStream	Flujo de entrada que lee desde un archivo
FileOutputStream	Flujo de salida que escribe a un archivo
FilterInputStream	Implementa InputStream
FilterOutputStream	Implementa OutputStream
InputStream	Clase abstracta que define la entrada de flujo
ObjectInputStream	Flujo de entrada para objetos
ObjectOutputStream	Flujo de salida para objetos
OutputStream	Clase abstracta que describe la salida de flujo
PipedInputStream	Línea de entrada
PipedOutputStream	Línea de salida
PrintStream	Flujo de salida que contiene **print()** y **println()**
PushbackInputStream	Flujo de entrada que permite que los bytes regresen al flujo
RandomAccessFile	Soporta la E/S de archivo de acceso directo
SequenceInputStream	Flujo de entrada que es una combinación de dos o más flujos de entrada, los cuales se leerán de manera secuencial, uno después de otro.

Tabla 10.1 Las clases de flujo de bytes.

teclado. **System.err** hace referencia al flujo de error estándar que, como opción predeterminada, también es la consola. Sin embargo, estos flujos pueden redirigirse a cualquier dispositivo de E/S compatible.

System.in es un objeto de tipo **InputStream**, mientras que **System.out** y **System.err** son objetos de tipo **PrintStream**. Se trata de flujos de bytes, si bien suelen emplearse para leer caracteres de la consola y escribirlos en ella. La razón de que sean flujos de bytes y no de caracteres es que los flujos predefinidos fueron parte de la especificación original de Java, la cual no incluía los flujos de caracteres. Como verá más adelante, si lo desea, es posible envolver éstos dentro de flujos de caracteres.

Clase de flujo de caracteres	Significado
BufferedReader	Flujo de caracteres de entrada almacenado en búfer
BufferedWriter	Flujo de caracteres de salida almacenado en búfer
CharArrayReader	Flujo de entrada que lee a partir de una matriz de caracteres
CharArrayWriter	Flujo de salida que escribe a una matriz de caracteres
FileReader	Flujo de entrada que lee a partir de un archivo
FileWriter	Flujo de salida que escribe a un archivo
FilterReader	Lector filtrado
FilterWriter	Escritura filtrada
InputStreamReader	Flujo de entrada que traduce bytes en caracteres
LineNumberReader	Flujo de entrada que cuenta líneas
OutputStreamWriter	Flujo de salida que traduce caracteres en bytes
PipedReader	Línea de entrada
PipedWriter	Línea de salida
PrintWriter	Flujo de salida que contiene **print()** y **println()**
PushbackReader	Flujo de entrada que permite que los caracteres se regresen al flujo de entrada
Reader	Clase abstracta que describe la entrada de flujo de caracteres
StringReader	Flujo de entrada que lee a partir de una cadena
StringWriter	Flujo de salida que escribe a una cadena
Writer	Clase abstracta que describe la salida de flujo de caracteres

Tabla 10.2 Clases de E/S de flujo de caracteres.

Comprobación de avance

1. ¿Qué es un flujo?

2. ¿Qué tipos de flujos define Java?

3. ¿Cuáles son los flujos integrados?

1. Un flujo es una abstracción que produce o consume información.
2. Java define flujos de bytes y de caracteres.
3. **System.in**, **System.out** y **System.err**.

HABILIDAD FUNDAMENTAL 10.6

Uso de los flujos de bytes

Empezaremos nuestro examen de la E/S de Java con los flujos de bytes. Como ya se explicó, en la parte superior de la jerarquía de flujos de bytes se encuentran las clases **InputStream** y **OutputStream**. En la tabla 10.3 se muestran los métodos de **InputStream** y en la 10.4 los de **OutputStream**. En general, los métodos en ambas clases pueden lanzar una **IOException** cuando un error ocurre. Los métodos definidos por esas dos clases abstractas están disponibles para todas sus subclases. Por lo tanto, forman un conjunto mínimo de funciones de E/S que todos los flujos de bytes tendrán.

Lectura de la entrada de la consola

Originalmente, la única manera de realizar la entrada en la consola era mediante el uso de un flujo de bytes. De hecho, buena parte del código de Java aún utiliza exclusivamente los flujos de bytes. Hoy día, puede usar flujos de bytes o de caracteres. Para el caso del código comercial, el método preferido

Método	Descripción
int available()	Regresa el número de bytes de la entrada que está disponible para lectura.
void close()	Cierra la fuente de entrada. Intentos adicionales de lectura causarán una **IOException**.
void mark(int *numBytes*)	Coloca una marca en el punto actual en el flujo de entrada que seguirá siendo válido hasta que *numBytes* de bytes sean leídos.
boolean markSupported()	Regresa **true** si **mark()/reset()** es soportado por el flujo que invoca.
int read()	Regresa una representación entera del siguiente byte disponible en la entrada. Se regresa –1 cuando se encuentra el final del archivo.
int read(byte *búfer*[])	Trata de leer hasta *búfer.longitud* bytes en búfer y regresa el número real de bytes que se leyó con éxito. Se regresa –1 cuando se encuentra el final del archivo.
int read(byte *búfer*[], int *desplazamiento*, int *numBytes*)	Trata de leer hasta *numBytes* bytes en búfer empezando en *búfer* [*desplazamiento*] y regresando el número de bytes que se leyó con éxito. Se regresa –1 cuando se encuentra el final del archivo.
void reset()	Restablece el puntero de entrada a la marca previamente establecida.
long skip(long *numBytes*)	Ignora (es decir, omite) *numBytes* bytes de entrada regresando el número de bytes que se ignora.

Tabla 10.3 Los métodos definidos por **InputStream**.

Método	Descripción
void close()	Cierra el flujo de salida. Los intentos adicionales de escribir generarán una **IOException**.
void flush()	Finaliza el estado de salida para que se limpien los búfers. Es decir, limpia los búfers de salida.
void write(int *b*)	Escribe un solo byte a un flujo de salida. Note que el parámetro es un **int**, lo que le permite llamar a **write** con expresiones, sin tener que moldearlos a **byte**.
void write(byte *búfer*[])	Escribe una matriz completa de bytes a un flujo de salida.
void write(byte *búfer*[], int *desplazamiento*, int *numBytes*)	Escribe un subrango de *numBytes* bytes desde el *búfer* de la matriz, empezando en *búfer[desplazamiento]*.

Tabla 10.4 Los métodos definidos por **OutputStream**.

para la lectura de la entrada en la consola es usar un flujo de caracteres. Al hacerlo así se logra que su programa resulte más fácil de internacionalizar y de mantener. También es más conveniente operar directamente con caracteres en lugar de convertir una y otra vez de caracteres a bytes o viceversa. Sin embargo, para programas de ejemplo, programas simples de utilería para uso personal y aplicaciones que tratan con la entrada simple del teclado, el uso de los flujos de bytes resulta aceptable. Por ello, a continuación se examinará la consola de E/S que utiliza flujos de bytes.

Debido a que **System.in** es una instancia de **InputStream**, tendrá acceso automáticamente a los métodos definidos por **InputStream**. Por desgracia, **InputStream** sólo define un método de entrada, **read()**, el cual lee bytes. Hay tres versiones de **read()**, mismas que se muestran a continuación:

int read( ) throws IOException

int read(byte *datos*[ ]) throws IOException

int read(byte *datos*[ ], int *inicio*, int *max*)throws IOException

En el módulo 3 se explicó cómo usar la primera versión de **read()** para leer un solo carácter del teclado (con **System.in**). Regresa –1 cuando se llega al final del flujo. La segunda versión lee bytes del flujo de entrada y los coloca en *datos* hasta que la matriz se llene, se alcance el final del flujo o un error ocurra. Regresa el número de bytes leídos, o –1 cuando se encuentra el final del flujo. La tercera versión lee la entrada de *datos* a partir de la ubicación especificada por *inicio*. Se almacena un máximo de *max* bytes; devuelve el número de bytes leídos, o –1 cuando se alcanza el final del flujo. Todos lanzan una **IOException** cuando un error ocurre. Cuando se lee de **System.in**, se genera, al oprimir ENTER, una condición de final de flujo.

He aquí un programa que demuestra la lectura de una matriz de bytes desde **System.in**.

```
// Lee una matriz de bytes del teclado.

import java.io.*;

class LeeBytes {
  public static void main(String args[])
    throws IOException {
      byte datos[] = new byte[10];

      System.out.println("Escriba algunos caracteres.");
      System.in.read(datos);
      System.out.print("Usted escribió: ");
      for(int i=0; i < datos.length; i++)
        System.out.print((char) datos[i]);
  }
}
```

Lee una matriz de bytes del teclado.

He aquí una ejecución de ejemplo:

```
Escriba algunos caracteres.
Lee bytes
Usted escribió: Lee bytes
```

Escritura de salida en la consola

Como en el caso de la entrada de la consola, Java originalmente proporcionaba sólo flujo de bytes para la salida de la consola. En Java 1.1 se agregaron los flujos de caracteres. Para el código más portable se recomiendan los flujos de caracteres. Sin embargo, debido a que **System.out** es un flujo de bytes, aún se emplea ampliamente la salida de bytes a la consola. En realidad, todos los programas que se han visto hasta el momento en este libro la han usado. Se examinarán a continuación.

La salida de la consola se logra más fácilmente con **print()** y **println()**, con los que ya está familiarizado. Estos métodos se definen con la clase **PrintStream** (que es el tipo de objeto al que hace referencia **System.out**). Aunque **System.out** es un flujo de bytes, aún resulta aceptable usar este flujo para salida simple de la consola.

Debido a que **PrintStream** es un flujo de salida derivado de **OutputStream**, implementa también el método de bajo nivel **write()**. Por consiguiente, es posible escribir en la consola empleando **write()**. Aquí se muestra la forma más simple de **write()** definida por **PrintStream**:

void write(in *valbyte*) throws IOException

Este método escribe el byte especificado por *valbyte* al archivo. Aunque *valByte* está declarado como un entero, sólo se escriben los 8 bites de orden inferior. He aquí un ejemplo breve que usa **write()** para dar salida al carácter "X" seguido por una nueva línea:

```
// Demuestra System.out.write().
class WriteDemo {
  public static void main(String args[]) {
    int b;

    b = 'X';
    System.out.write(b); <-- Escribe un byte en la pantalla.
    System.out.write('\n');
  }
}
```

No usará con frecuencia **write()** para realizar salida de la consola (aunque resultaría útil en algunas situaciones), porque **print()** y **println()** son sustancialmente más fáciles de emplear.

A partir de J2SE 5, **PrintStream** proporciona dos métodos de salida adicionales: **printf()** y **format()**. Ambos le proporcionan control detallado sobre el formato preciso de los datos a los que está dando salida. Por ejemplo, puede especificar el número de lugares decimales desplegados, un ancho de campo mínimo o el formato de un valor negativo. Aunque no usaremos estos métodos en los ejemplos, éstos constituyen características que querrá revisar a medida que adquiera más conocimientos de Java.

Comprobación de avance

1. ¿Qué método se usa para leer un byte de **System.in**?
2. Aparte de **Print()** y **println()**, ¿qué método puede usarse para escribir a **System.out**?

HABILIDAD FUNDAMENTAL 10.7

Lectura y escritura de archivo empleando flujos de byte

Java proporciona varias clases y métodos que le permiten leer y escribir archivos. Por supuesto, los tipos más comunes de archivos son los de disco. En Java, todos los archivos están orientados a bytes. En este sentido, Java proporciona métodos para leer y escribir bytes en archivos. De ahí que la lectura y escritura de archivos empleando flujos de bytes sea muy común. Sin embargo, Java le permite

1. Para leer un byte, llame a **read()**.
2. Puede escribir de **System.out** llamando a **write()**.

envolver un flujo de archivo orientado a bytes dentro de un objeto de caracteres, lo que se muestra más adelante en este módulo.

Para crear un flujo de bytes vinculado con un archivo, debe usar **FileInputStream** o **FileOutputStream**. Para abrir un archivo, simplemente cree un objeto de una de esas clases especificando el nombre del archivo como argumento para el constructor. Una vez que el archivo esté abierto, puede leerlo o escribir en él.

Obtención de entrada de un archivo

Un archivo se abre para entrada mediante la creación de un objeto **FileInputStream**. He aquí el constructor más usado:

FileInputStream(String *nombreArchivo*) throws **FileNotFoundException**

Aquí, *nombreArchivo* especifica el nombre del archivo que desea abrir. Si el archivo no existe, entonces se lanza **FileNotFoundException**.

Para leer un archivo, puede usar **read()**. Ésta es la versión que usaremos:

int read() throws IOException

Cada vez que es llamado, **read()** lee un solo byte del archivo y lo regresa como un valor de entero; regresa –1 cuando se encuentra el final del archivo, y lanza una **IOException** cuando un error ocurre. Por lo tanto, esta versión de **read()** es la misma que la que se utiliza para leer de la consola.

Cuando haya terminado con un archivo, debe cerrarlo llamando a **close()**. Aquí se muestra su forma general:

void close( ) throws IOException

Al cerrar un archivo se liberan recursos del sistema asignados al archivo, lo que permite que otro archivo los utilice.

El siguiente programa usa **read()** para obtener entrada y despliega el contenido de un archivo de texto, cuyo nombre se especifica como un argumento de línea de comandos. Observe los bloques **try/catch** que manejan los dos errores que podrían ocurrir al emplear este programa: el archivo especificado no se encuentra o el usuario olvidó incluir el nombre del archivo. Puede usar este mismo método siempre que use argumentos de línea de comandos.

```
/* Despliega un archivo de texto.

   Para usar este programa, especifique el nombre
   del archivo que quiere ver.
   Por ejemplo, para ver un archivo llamado PRUEBA.TXT,
   use la siguiente línea de comandos.
```

```
   java MostrarArchivo PRUEBA.TXT
*/

import java.io.*;

class MostrarArchivo {
  public static void main(String args[])
    throws IOException
  {
    int i;
    FileInputStream aren;

    try {
      aren = new FileInputStream(args[0]);
    } catch(FileNotFoundException exc) {
      System.out.println("Archivo no encontrado");
      return;
    } catch(ArrayIndexOutOfBoundsException exc) {
      System.out.println("Uso: MostrarArchivo Archivo");
      return;
    }

    // lee bytes hasta el final del archivo
    do {
      i = aren.read();
      if(i != -1) System.out.print((char) i);
    } while(i != -1);

    aren.close();
  }
}
```

Lee del archivo .

Cuando i es igual a –1, se ha alcanzado el final del archivo.

Pregunte al experto

P: **Noté que** read( ) **regresa –1 cuando se alcanza el final del archivo, pero que no tiene un valor de regreso especial para un error de archivo. ¿Por qué?**

R: En Java, las excepciones manejan los errores. Por lo tanto, si **read()**, o cualquier otro método de E/S, regresa un valor, significa que se no ha presentado un error. Se trata de una manera mucho más limpia que la de manejar los errores de E/S usando códigos de error especiales.

Escritura en un archivo

Para abrir un archivo para salida, debe crear un objeto de **FileOutputStream**. He aquí dos de los constructores de uso más común:

FileOutputStream(String *nombreArchivo*) throws FileNotFoundException

FileOutputStream(String *nombreArchivo*, boolean *adjunto*)
throws FileNotFoundException

Si no puede crearse el archivo, entonces se lanza la excepción **FileNotFoundException**. En la primera forma, cuando el archivo de salida está abierto, se destruye cualquier archivo existente que tenga el mismo nombre. En la segunda forma, si *adjuntos* es **true**, entonces la salida se adjunta al final del archivo. De otra manera, se sobrescribe el archivo.

Para escribir en un archivo, usará el método **write( )**. Aquí se muestra su forma más simple:

void write(in *valbyte*) throws IOException

Este método escribe en el archivo el byte especificado por *valbyte*. Aunque *valbyte* se declare como entero, sólo se escribirán en el archivo los 8 bits de orden inferior. Si un error ocurre durante la escritura, se lanza una **IOException**.

Como tal vez ya lo sepa, cuando se realiza la salida de un archivo, ésta no se escribe de inmediato en el dispositivo físico. En cambio, la salida se almacena en búfer hasta que se tiene un fragmento considerable de datos que puedan ser escritos de una sola vez, lo que mejora la eficiencia del sistema. Por ejemplo, los archivos de disco están organizados en sectores, que podrían tener 128 bytes o más de largo. La salida suele almacenarse hasta que pueda escribirse un sector completo a la vez. Sin embargo, si quiere que los datos se escriban en el dispositivo físico, esté lleno el búfer o no, puede llamar a **flush( )**, el cual se muestra a continuación:

void flush( ) throws IOException

Se lanza una excepción en caso de falla.

Una vez que haya terminado con un archivo de salida, debe cerrarlo usando **close( )**, el cual se muestra aquí:

void close( ) throws IOException

Al cerrar un archivo se liberan recursos del sistema asignados al archivo, permitiendo que otros archivos los utilicen. Asimismo, asegura que se escriba cualquier salida que permanezca en el búfer del disco.

El siguiente ejemplo copia un archivo de texto. Los nombres de los archivos de origen y destino se especifican en la línea de comandos.

```
/* Copia un archivo de texto.

   Para usar este programa, especifique el nombre
   de los archivos de origen y de destino.
   Por ejemplo, para copiar un archivo llamado PRIMERO.TXT
   en un archivo llamado SEGUNDO.TXT, use la siguiente
   línea de comandos:

   java CopiarArchivo PRIMERO.TXT SEGUNDO.TXT
*/

import java.io.*;

class CopiarArchivo {
  public static void main(String args[])
    throws IOException
  {
    int i;
    FileInputStream aren;
    FileOutputStream arsal;

    try {
      // abre archivo de entrada
      try {
        aren = new FileInputStream(args[0]);
      } catch(FileNotFoundException exc) {
        System.out.println("Archivo de entrada no encontrado");
        return;
      }

      // abre archivo de salida
      try {
        arsal = new FileOutputStream(args[1]);
      } catch(FileNotFoundException exc) {
        System.out.println("Error al abrir el archivo de salida");
        return;
      }
    } catch(ArrayIndexOutOfBoundsException exc) {
      System.out.println("Uso: CopiarArchivo De A");
      return;
    }
```

```
    // Copia el archivo
    try {
      do {
        i = aren.read();  <-- Lee bytes de una línea
        if(i != -1) arsal.write(i);  <-- y los escribe en otra.
      } while(i != -1);
    } catch(IOException exc) {
      System.out.println("Error en archivo");
    }

    aren.close();
    arsal.close();
  }
}
```

Comprobación de avance

1. ¿Qué regresa **read()** cuando se alcanza el final del archivo?
2. ¿Qué función realiza **flush()**?

HABILIDAD FUNDAMENTAL 10.8

Lectura y escritura de datos binarios

Hasta el momento hemos estado leyendo y escribiendo bytes que contienen caracteres ASCII, pero es posible (y común) leer y escribir otro tipos de datos. Por ejemplo, tal vez quiera crear un archivo que contenga tipos de enteros **int**, **double** o **short**. Para leer y escribir valores binarios de los tipos primitivos de Java, deberá usar **DataInputStream** y **DataOutputStream**.

DataOutputStream implementa la interfaz **DataOutput**. Esta interfaz define métodos que escriben todos los tipos primitivos de Java en un archivo. Es importante comprender que estos datos se escriben usando su formato interno y binario, y no una forma de texto legible para los seres humanos. Los métodos de salida de uso más común para los tipos primitivos de Java se muestran en la tabla 10.5. Cada una lanza una **IOException** en caso de que haya una falla.

He aquí el constructor de **DataOutputStream**. Tome en cuenta que está construido sobre una instancia de **OutputStream**.

DataOutputStream(OutputStream *flujo Salida*)

1. **read** regresa a un valor –1, cuando se encuentra el final del archivo.
2. Una llamada a **flush** origina que cualquier salida en búfer se escriba físicamente en el dispositivo de almacenamiento.

Método de salida	Propósito
void writeBoolean (boolean val)	Escribe el valor **boolean** especificado por *val*.
void writeByte(int *val*)	Escribe el byte de orden inferior especificado por *val*.
void writeChar(int *val*)	Escribe el valor especificado por *val* como un carácter.
void writeDouble(double *val*)	Escribe el valor **double** especificado por *val*.
void writeFloat(float *val*)	Escribe el valor **float** especificado por *val*.
void writeInt(int *val*)	Escribe el valor **int** especificado por *val*.
void writeLong(long *val*)	Escribe el valor **long** especificado por *val*.
void writeShort(int *val*)	Escribe el valor especificado por *val* como **short**.

Tabla 10.5 Métodos de salida de uso común definidos por **DataOutputStream**.

Aquí, *flujoSalida* es el flujo al que se escriben los datos. Para escribir salida en un archivo, puede usar el objeto creado por **FileOutputStream** para este parámetro.

DataInputStream implementa la interfaz **DataInput**, que proporciona métodos para leer todos los tipos primitivos de Java. Estos métodos se muestran en la tabla 10.6 y cada uno puede lanzar una **IOException**. **DataInputStream** usa una instancia de **InputStream** como base, la cual superpone con métodos que leen los diversos tipos de datos de Java. Recuerde que **DataInputStream** lee datos en su formato binario, no es su forma legible para los seres humanos. El constructor para **DataInputStream** se muestra a continuación.

DataInputStream(InputStream *flujoEntrada*)

Método de salida	Propósito
boolean readBoolean()	Lee un valor **boolean**.
byte readByte()	Lee un valor **byte**.
char readChar()	Lee un valor **char**.
double readDouble()	Lee un valor **double**.
float readFloat()	Lee un valor **float**.
int readInt()	Lee un valor **int**.
long readLong()	Lee un valor **long**.
short readShort()	Lee un valor **short**.

Tabla 10.6 Métodos de entrada de uso común definidos por **DataInputStream**.

Aquí, *flujo Entrada* es el flujo que está vinculado con la instancia de **DataInputStream** que se está creando. Para leer la entrada de un archivo, puede usar el objeto creado por **FileInputStream** para este parámetro.

He aquí un programa que demuestra **DataOutputStream** y **DataInputStream**. Escribe y luego lee varios tipos de datos de un archivo a otro.

```
// Escribe y luego lee datos binarios.
import java.io.*;

class LEDatos {
  public static void main(String args[])
    throws IOException {

    DataOutputStream datosSal;
    DataInputStream datosEn;

    int i = 10;
    double d = 1023.56;
    boolean b = true;

    try {
      datosSal = new
          DataOutputStream(new FileOutputStream("datos de prueba"));
    }
    catch(IOException exc) {
      System.out.println("No se puede abrir archivo.");
      return;
    }

    try {
      System.out.println("Escribiendo " + i);
      datosSal.writeInt(i);

      System.out.println("Escribiendo " + d);
      datosSal.writeDouble(d);

      System.out.println("Escribiendo " + b);
      datosSal.writeBoolean(b);

      System.out.println("Escribiendo " + 12.2 * 7.4);
      datosSal.writeDouble(12.2 * 7.4);

    }
    catch(IOException exc) {
      System.out.println("Error de escritura.");
    }
```

Escribe datos binarios.

```
    datosSal.close();

    System.out.println();

    // Ahora lee.
    try {
      datosEn = new
          DataInputStream(new FileInputStream("datos de prueba"));
    }
    catch(IOException exc) {
      System.out.println("No puede abrir el archivo.");
      return;
    }

    try {
      i = datosEn.readInt();
      System.out.println("Leyendo " + i);

      d = datosEn.readDouble();
      System.out.println("Leyendo " + d);

      b = datosEn.readBoolean();
      System.out.println("Leyendo " + b);

      d = datosEn.readDouble();
      System.out.println("Leyendo " + d);
    }
    catch(IOException exc) {
      System.out.println("Error de lectura.");
    }

    datosEn.close();
  }
}
```

Lee datos binarios.

Ésta es la salida de este programa:

```
Escribiendo 10
Escribiendo 1023.56
Escribiendo true
Escribiendo 90.28

Leyendo 10
Leyendo 1023.56
Leyendo true
Leyendo 90.28
```

Comprobación de avance

1. Para escribir datos binarios, ¿qué tipo de flujo debe usar?
2. ¿A qué método llama para escribir un tipo **double**?
3. ¿A qué método llama para leer un tipo **short**?

Proyecto 10.1 Una utilería de comparación de archivos

CompArchivos.java

Este proyecto desarrolla una utilería de comparación de archivos simple pero útil. Funciona al abrir dos archivos que se compararán para luego leer y comparar cada uno de los conjuntos correspondientes de bytes. Si se encuentra alguna discrepancia, entonces los archivos son diferentes. Si se alcanza el final de cada archivo al mismo tiempo y no se encuentran discrepancias, los archivos son iguales.

Paso a paso

1. Cree un archivo llamado **CompArchivos.java**.
2. En **CompArchivos.java**, agregue el siguiente programa:

```
/*
   Proyecto 10.1

   Compara dos archivos.

   Para usar este programa, especifique los nombres
   de los archivos que se compararán
   en la línea de comandos.

   java CompArchivo PRIMERO.TXT SEGUNDO.TXT
*/

import java.io.*;

class CompArchivos {
  public static void main(String args[])
    throws IOException
  {
```

1. Para escribir datos, debe usar **DataOutputStream**.
2. Para escribir un tipo **double**, debe llamar a **writeDouble()**.
3. Para leer un tipo short, debe llamar a **readShort()**.

```
    int i=0, j=0;
    FileInputStream a1;
    FileInputStream a2;

    try {
      // abre el primer archivo
      try {
        a1 = new FileInputStream(args[0]);
      } catch(FileNotFoundException exc) {
        System.out.println(args[0] + " No se encuentra el archivo");
        return;
      }

      // abre el segundo archivo
      try {
        a2 = new FileInputStream(args[1]);
      } catch(FileNotFoundException exc) {
        System.out.println(args[1] + " No se encuentra el archivo");
        return;
      }
    } catch(ArrayIndexOutOfBoundsException exc) {
      System.out.println("Uso: CompArchivo a1 a2");
      return;
    }

    // Compara archivos
    try {
      do {
        i = a1.read();
        j = a2.read();
        if(i != j) break;
      } while(i != -1 && j != -1);
    } catch(IOException exc) {
      System.out.println("Error de archivo");
    }
    if(i != j)
      System.out.println("Archivos diferentes.");
    else
      System.out.println("Los archivos son iguales.");

    a1.close();
    a2.close();
  }
}
```

(continúa)

3. Para probar **CompArchivos**, primero copie **CompArchivos.java** en un archivo llamado **temp**. Luego pruebe esta línea de comandos:

```
java CompArchivos CompArchivos.java temp
```

El programa reportará que los archivos son iguales. A continuación, compare **CompArchivos.java** con **CopiarArchivo.java** (que se mostró antes) usando la línea de comandos:

```
java CompArchivos CompArchivos.java CopiarArchivo.java
```

Estos archivos son diferentes, así que **CompArchivos** lo reportará.

4. Por su cuenta, trate de mejorar **CompArchivos** mediante varias opciones. Por ejemplo, agregue una opción que ignore las mayúsculas. Otra idea es hacer que **CompArchivos** despliegue la posición en la que los archivos son diferentes.

HABILIDAD FUNDAMENTAL 10.9

Archivos de acceso directo

Hasta este punto, hemos estado usando *archivos secuenciales*, que son aquellos que se pueden acceder de manera estrictamente secuencial, byte tras byte. Sin embargo, Java también le permite acceder al contenido de un archivo de manera directa. Para ello, usará **RandomAccessFile**, que encapsula un archivo de acceso directo. **RandomAccessFile** no deriva de **InputStream** ni de **OutputStream**, sino que implementa las interfaces **DataInput** y **DataOutput**, que definen los métodos básicos de E/S. También soporta la solicitud de posicionamiento (es decir, puede colocar el *puntero de archivo* dentro del archivo). A continuación se muestra el constructor que estaremos usando:

RandomAccessFile(String *nombreArchivo*, String *acceso*)
throws FileNotFoundException

Aquí, el nombre del archivo se pasa en *nombre archivo,* mientras que *acceso* determina el tipo de acceso de archivo que se permite. Si es "r", el archivo puede leerse pero no escribirse. Si es "rw", el archivo está abierto en el modo de lectoescritura.

El método **seek()**, que se muestra aquí, se usa para establecer la posición actual del puntero dentro del archivo.

void seek(long *nuevaPos*) throws IOException

Aquí, *nuevaPos* especifica la nueva posición, en bytes, del puntero desde el principio del archivo. Después de una llamada a **seek()**, la siguiente operación de lectoescritura ocurrirá en la nueva posición de archivo.

RandomAccessFile implementa los métodos **read()** y **write()**. También implementa las interfaces **DataInput** y **DataOutput**, lo que significa que los métodos, como **readInt()** y **writeDouble()**, están disponibles para leer y escribir los tipos primitivos.

Aquí está un ejemplo que muestra accesos directos I/0. Escribe 6 doubles a un archivo y luego los lee de regreso en orden no secuencial.

```
// Demuestra los archivos de acceso directo.
import java.io.*;

class AccesoDirectoDemo {
  public static void main(String args[])
    throws IOException {

    double datos[] = { 19.4, 10.1, 123.54, 33.0, 87.9, 74.25 };
    double d;
    RandomAccessFile aad;

    try {
      aad = new RandomAccessFile("directo.dat", "rw");
    }
    catch(FileNotFoundException exc) {
      System.out.println("No puede abrir el archivo.");
      return ;
    }

    // escribe los valores en el archivo.
    for(int i=0; i < datos.length; i++) {
      try {
        aad.writeDouble(datos[i]);
      }
      catch(IOException exc) {
        System.out.println("Error al escribir el archivo.");
        return ;
      }
    }

    try {
      // Ahora, vuelve a leer valores específicos
      aad.seek(0); // busca el primer double
      d = aad.readDouble();
      System.out.println("El primer valor es " + d);

      aad.seek(8); // busca el segundo double
      d = aad.readDouble();
      System.out.println("El segundo valor es " + d);

      aad.seek(8 * 3); // busca el cuarto double
      d = aad.readDouble();
      System.out.println("El cuarto valor es " + d);
```

Abre un archivo de acceso directo.

Use **seek()** para establecer el puntero de archivo.

```
      System.out.println();

      // Ahora lee todos los demás valores.
      System.out.println("Aquí están todos los demás valores: ");
      for(int i=0; i < datos.length; i+=2) {
        aad.seek(8 * i); // busca el iésimo double
        d = aad.readDouble();
        System.out.print(d + " ");
      }
    }
    catch(IOException exc) {
      System.out.println("Error buscando o leyendo.");
    }

    aad.close();
  }
}
```

Ésta es la salida de este programa:

```
El primer valor es 19.4
El segundo valor es 10.1
El cuarto valor es 33.0

Éstos son todos los demás valores:
19.4 123.54 87.9
```

Observe cómo se localiza cada valor. Como cada valor double es de 8 bytes, cada valor inicia en un límite de 8 bytes. Por lo tanto, el primer valor se localiza en cero, el segundo empieza en el byte 8, el tercero en el 16, etc. Así, para leer el cuarto valor, el programa busca la ubicación 24.

Comprobación de avance

1. ¿Qué clase debe usar para crear un archivo de acceso directo?

2. ¿Cómo se coloca el puntero de archivo?

1. Para crear un archivo de acceso directo, debe usar **RandomAccessFile**.
2. Para colocar el puntero de archivo, debe usar **seek()**.

HABILIDAD FUNDAMENTAL 10.10

Uso de los flujos de caracteres de Java

Como se ha mostrado en las secciones anteriores, los flujos de bytes de Java son muy poderosos y flexibles. Sin embargo, no son la manera ideal de manejar E/S de caracteres. Para este fin, Java define las clases de flujo de caracteres. En la parte superior de la jerarquía de flujo de caracteres se encuentran las clases abstractas **Reader** y **Writer**. En la tabla 10.7 se muestran los métodos de **Reader** y en la tabla 10.8 se muestran los de **Writer**. Todos los métodos pueden lanzar una **IOException** cuando haya un error. Los métodos definidos por estas dos clases abstractas se encuentran disponibles en todas sus subclases. En consecuencia, forman un conjunto mínimo de funciones de E/S que todos los flujos de caracteres tendrán.

Método	Descripción
abstract void close()	Cierra la fuente de entrada. Intentos adicionales de lectura generarán una **IOException**.
void mark(in *numCars*)	Coloca una marca en el punto actual en el flujo de entrada que seguirá siendo válido hasta que se lean *numCars*.
boolean markSupported()	Regresa **true** si **mark()/reset()** están soportados en este flujo.
int read()	Regresa una representación entera del siguiente carácter disponible a partir del flujo de entrada que invoca. Se regresa –1 cuando se encuentra el final del archivo.
int read(char *búfer*[])	Trata de leer hasta *búfer.longitud* en *búfer* y regresa el número real de caracteres que se leyeron con éxito. Se regresa –1 cuando se encuentra el final del archivo.
abstract int reader (char *búfer*[], int *desplazamiento*, int *numCars*)	Trata de leer hasta caracteres *numCars* en *búfer* a partir de *búfer* [*desplazamiento*], regresando el número de caracteres que se leyó con éxito. Se regresa –1 cuando se encuentra el final del archivo.
int read(CharBuffer *búfer*)	Trata de llenar el búfer especificado por *búfer* regresando el número de caracteres que se leyó con éxito. Se regresa –1 cuando se encuentra el final del archivo. **CharBuffer** es una clase que encapsula una secuencia de caracteres, como una cadena.
boolean ready()	Regresa **true** si la siguiente solicitud de entrada no espera. De otra manera, regresa **false**.
void reset()	Restablece el puntero de entrada en la marca establecida previamente.
long skip(long *numCars*)	Omite *numCars* de entrada regresando el número de caracteres que se omitió.

Tabla 10.7 Los métodos definidos por **Reader**.

Método	Descripción
Writer append(char *car*) throws IOException	Añade *car* al final del flujo de salida que invoca. Devuelve una referencia al flujo que invoca. (Añadido en J2SE 5.)
Writer append(CharSequence *cars*) throws IOException	Añade *cars* al final del flujo de salida que invoca. Devuelve una referencia al flujo que invoca. **CharSequence** es una interfaz que define operaciones de sólo lectura en una secuencia de caracteres. (Añadido en J2SE 5.)
Writer append(CharSequence *cars*, int *inicio*, int *final*) *throws IOException*	Añade la secuencia de *cars* que empieza en *inicio* y se detiene con *final* al final del flujo de salida que invoca. Devuelve una referencia al flujo que invoca. **CharSequence** es una interfaz que define operaciones de sólo lectura en una secuencia de caracteres. (Añadido en J2SE 5.)
abstract void close()	Cierra el flujo de salida. Los intentos adicionales de escritura generarán una **IOException**.
abstract void flush()	Finaliza el estado de salida para que se limpie cualquier búfer de salida.
void write(int *car*)	Escribe un solo carácter al flujo de salida que invoca. Observe que el parámetro es un **int**, lo que le permite llamar a **write** con expresiones sin tener que moldearlas a **char**.
void write(char *búfer*[])	Escribe una matriz de caracteres completa en el flujo de salida que invoca.
abstract void write(char *búfer*[], int *desplazamiento*, int *numCars*)	Escribe un subrango de caracteres *numCars* del *búfer* de la matriz, empezando en búfer[*desplazamiento*] al flujo de salida que invoca.
void write(String *cad*)	Escriba *cad* en el flujo de salida que invoca.
void write(String *cad*, int *desplazamiento*, int *numCars*)	Escribe un subrango de caracteres *numCars* de la matriz *cad* empezando en el *desplazamiento* especificado.

Tabla 10.8 Los métodos definidos por **Writer**.

Entrada de consola empleando flujos de caracteres

En el caso del código que habrá de usarse internacionalmente, resulta mejor y más conveniente usar la entrada de la consola mediante flujos de caracteres de Java para leer caracteres del teclado que usar flujos de bytes. Sin embargo, como **System.in** es un flujo de bytes, necesitará envolver **System.in** dentro de algún tipo de **Reader**. La mejor clase para leer la entrada de la consola es **BufferedReader**, el cual soporta un flujo de entrada en búfer. Sin embargo, no puede construir una clase **BufferedReader** directamente de **System.in**. En cambio, debe primero convertirla en un flujo de caracteres. Para ello, usará **InputStreamReader**, que convierte bytes en caracteres. Para obtener un objeto de **InputStreamReader** que esté vinculado con **System.in**, use el siguiente constructor:

InputStreamReader(InputStream *FlujoEntrada*)

Como **System.in** hace referencia a un objeto de tipo **InputStream**, puede emplearse entonces para *FlujoEntrada*.

A continuación, con el objeto producido por **InputStreamReader**, construya una clase **BufferedReader** empleando el siguiente constructor:

BufferedReader(Reader *lectorEntrada*)

Aquí, *lectorEntrada* es el flujo que está vinculado con la instancia de **BufferedReader** que se está creando. Al poner todo junto, la siguiente línea de código crea un **BufferedReader** que está conectado al teclado:

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

Después de que esta instrucción se ejecuta, **br** se convierte en un flujo de carácter vinculado con la consola a través de **System.in.**

Lectura de caracteres

Los caracteres pueden leerse a partir de **System.in** empleando el método **read()** definido por **BufferedReader** de manera muy parecida a la forma en que se leen mediante los flujos de bytes. He aquí tres versiones de **read()** definidos por **BufferedReader**:

int read( ) throws IOException

int read(char *datos*[ ]) throws IOException

int read(char *datos*[ ]) int *inicio*, int *max*) throws IOException

La primera versión de **read()** lee un solo carácter de Unicode y regresa –1 cuando se alcanza el final del flujo. La segunda versión lee caracteres del flujo de entrada y los coloca en *datos* hasta que la matriz se llena, el final del archivo se alcanza o un error ocurre. Además, regresa el número de caracteres leídos o –1 al final del flujo. La tercera versión lee entrada de *datos* a partir de la ubicación especificada por *inicio*. Almacena un máximo de *max* caracteres y regresa el número de caracteres leídos o –1 cuando se alcanza el final del archivo. Todas lanzan una **IOException** cuando un error ocurre. Cuando se lee a partir de **System.in**, se genera, al oprimir ENTER, una condición de final de flujo.

El siguiente programa demuestra **read()** mediante la lectura de caracteres de la consola hasta que el usuario escribe un punto.

```
// Use un BufferedReader para leer caracteres de la consola.
import java.io.*;

class LeerCars {
```

```
  public static void main(String args[])
    throws IOException
  {
    char c;
    BufferedReader br = new
            BufferedReader(new
    InputStreamReader(System.in));

     System.out.println("Escriba caracteres; un punto para terminar.");

    // lee caracteres
    do {
      c = (char) br.read();
      System.out.println(c);
    } while(c != '.');
  }
}
```

Crea **BufferedReader** vinculado a **System.in**.

He aquí una ejecución de ejemplo.

```
Escriba caracteres; un punto para terminar.
Uno dos.
U
n
o

d
o
s
.
```

Lectura de cadenas

Para leer una cadena del teclado, debe usarse la versión de **readLine()** que es miembro de la clase **BufferedReader**. Ésta es su forma general:

String readLine( ) throws IOException

Regresa un objeto de **String** que contiene los caracteres leídos. Regresa null si se hace un intento de leer cuando se llega al final del flujo.

El siguiente programa demuestra **BufferedReader** y el método **readLine()**. El programa lee y despliega líneas de texto hasta que usted ingrese la palabra "alto".

```
// Lee una cadena de la consola usando un BufferedReader.
import java.io.*;
```

```
class LeeLíneas {
  public static void main(String args[])
    throws IOException
  {
    // crea un BufferedReader con System.in
    BufferedReader br = new BufferedReader(new
                            InputStreamReader(System.in));
    String cad;

    System.out.println("Escriba líneas de texto.");
    System.out.println("Escriba 'alto' para salir.");
    do {
      cad = br.readLine();
      System.out.println(cad);
    } while(!cad.equals("alto"));
  }
}
```

Use **readLine()** de **BufferedReader** para leer una línea de texto.

Salida de consola usando flujos de caracteres

Aunque todavía se permite usar **System.out** para escribir a la consola bajo Java, su uso se recomienda principalmente para fines de depuración o para programas de ejemplo como los encontrados en este libro. En el caso de programas reales, el método preferido para escribir en la consola cuando se usa Java es el flujo de **PrintWriter**. **PrintWriter** es una de las clases que están basadas en caracteres. Como se explicó, el uso de una clase basada en caracteres para la salida de consola facilita la internacionalización de su programa.

PrintWriter define varios constructores. A continuación se muestra el que utilizaremos:

PrintWriter(OutputStream *flujoSalida*, boolean *limpiaEnNuevaLínea*)

Aquí, *flujoSalida* es un objeto de tipo **OutputStream**, mientras que *limpiaEnNuevaLínea* controla si Java limpia el flujo de salida cada vez que se llama a un método **println()**. Si *limpiaEnNuevaLínea* es **true**, entonces tiene lugar una limpieza automática. Si es **false**, la limpieza no es automática.

PrintWriter soporta los métodos **print()** y **println()** para todos los tipos, incluyendo **Object**. Por lo tanto, puede usar estos métodos de la misma manera en la que se han empleado con **System.out**. Si un argumento no es un tipo primitivo, los métodos de **PrintWriter** llamarán al método **toString()** del objeto y luego imprimirán el resultado.

Para escribir a la consola usando **PrintWriter**, especifique **System.out** para el flujo de salida y limpie el flujo después de cada llamada a **println()**. Por ejemplo, esta línea de código crea una clase **PrintWriter** que está conectada con la salida de la consola.

```
PrintWriter pw = new PrintWriter(System.out, true)
```

La siguiente aplicación ilustra el uso de un **PrintWriter** para manejar la salida de la consola.

Cree un vínculo entre **PrintWriter** y **System.out**.

```
// Demuestra PrintWriter.
import java.io.*;

public class PrintWriterDemo {
  public static void main(String args[]) {
    PrintWriter pw = new PrintWriter(System.out, true);
    int i = 10;
    double d = 123.65;

    pw.println("Uso de una PrintWriter.");
    pw.println(i);
    pw.println(d);

    pw.println(i + " + " + d + " es " + (i+d));
  }
}
```

La salida de este programa es:

```
Uso de una PrintWriter.
10
123.65
10 + 123.65 es 133.65
```

Recuerde que no hay nada malo en utilizar **System.out** para escribir salida de texto simple a la consola cuando está aprendiendo a usar Java o está depurando su programa. Sin embargo, el uso de **PrintWriter** hará que sus aplicaciones reales sean más fáciles de internacionalizar. Debido a que no hay ventajas en el uso de **PrintWriter** en los programas de ejemplo que se muestran en este libro, por razones de conveniencia seguiremos usando **System.out** para escribir en la consola.

Comprobación de avance

1. ¿Qué clases se encuentran en la parte superior de las clases de flujo de caracteres?
2. Para leer a partir de la consola, ¿qué tipo de lector deberá abrir?
3. Para escribir en la consola, ¿qué tipo de escritor deberá abrir?

1. En la parte superior de estas clases de flujo de caracteres están **Reader** y **Writer**.
2. Para leer desde la consola, abra **BufferedReader**.
3. Para escribir en la consola, abra **PrintWriter**.

HABILIDAD FUNDAMENTAL 10.11

E/S de archivo empleando flujos de caracteres

Aunque el manejo de archivos orientado a bytes es el más común, es posible usar flujos de caracteres para este fin. La ventaja de los flujos de caracteres es que operan directamente en los caracteres de Unicode. Por lo tanto, si quiere almacenar texto de Unicode, con toda seguridad los flujos de caracteres representan su mejor opción. En general, para realizar E/S de archivo de caracteres, deberá usar las clases **FileReader** y **FileWriter**.

Uso de FileWriter

FileWriter crea una clase **Writer** que escribe archivos. A continuación se muestran sus constructores de uso más común:

FileWriter(String *nombeArchivo*) throws IOException

FileWriter(String *nombeArchivo*, boolean *adjunto*) throws IOException

Aquí, *nombreArchivo* es el nombre de la ruta completa de un archivo. Si *adjunto* es **true**, entonces la salida se adjunta al final del archivo. De otra manera, el archivo es sobreescrito. Ambos constructores lanzan una **IOException** si una falla ocurre. **FileWriter** se deriva de **OutputStreamWriter** y **Writer**; así que tiene acceso a los métodos definidos por dichas clases.

He aquí una utilería simple de teclado a disco que lee líneas de texto introducidas en el teclado, las cuales ingresa en un archivo llamado "prueba.txt". El texto se lee hasta que el usuario ingresa la palabra "alto". Se usa una clase **FileWriter** para dar salida al archivo.

```
/* Una utilería simple de teclado a disco
   que demuestra una clase FileWriter. */

import java.io.*;

class TAD {
  public static void main(String args[])
  throws IOException {

    String cad;
    FileWriter fw;
    BufferedReader br =
            new BufferedReader(
                  new InputStreamReader(System.in));
```

```
    try {
      fw = new FileWriter("prueba.txt");
    }
    catch(IOException exc) {
      System.out.println("No puede abrir el archivo.");
      return ;
    }

    System.out.println("Escriba texto ('alto' para salir).");
    do {
      System.out.print(": ");
      cad = br.readLine();

      if(cad.compareTo("alto") == 0) break;

      cad = cad + "\r\n"; // agrega nueva línea
      fw.write(cad);
    } while(cad.compareTo("alto") != 0);

    fw.close();
  }
}
```

Crea un **FileWriter**.

Escribe cadenas en el archivo.

Uso de FileReader

La clase **FileReader** crea una clase **Reader** que se usa para leer el contenido de un archivo. El siguiente es su constructor de uso más común:

FileReader(String *nombreArchivo*) throws FileNotFoundException

Aquí, *nombreArchivo* es el nombre de la ruta completa de un archivo. En caso de que el archivo no exista, lanza una **FileNotFoundException**. **FileReader** se deriva de **InputStreamReader** y **Reader**; por consiguiente, tiene acceso a los métodos definidos por estas clases.

El siguiente programa crea una utilería sencilla de disco a pantalla que lee un archivo de texto llamado "prueba.txt" y despliega el contenido de la pantalla. Es el complemento de la utilería de teclado a disco que se mostró en la sección anterior.

```
/* Una utilería sencilla de disco a pantalla
   que demuestra una clase FileReader. */

import java.io.*;

class DAP {
  public static void main(String args[]) throws Exception {
```

```
    FileReader fr = new FileReader("prueba.txt");
    BufferedReader br = new BufferedReader(fr);
    String s;

    while((s = br.readLine()) != null) {
      System.out.println(s);
    }

    fr.close();
  }
}
```

Lee líneas del archivo y los despliega en la pantalla.

En este ejemplo, observe que el **FileReader** está envuelto en una **BufferedReader**. Esto le brinda acceso a **readLine()**.

Pregunte al experto

P: **He oído que se agregó recientemente un nuevo sistema de E/S. ¿Puede hablarme de él?**

R: En 2002, J2SE 1.4 se agregó una nueva manera de manejar operaciones de E/S llamada la *nueva API de E/S*, la cual constituye una de las adiciones más interesantes de Sun a la versión 1.4 porque soporta un método basado en canal para las operaciones de E/S. Las nuevas clases de E/S están contenidas en **java.nio** y en sus paquetes subordinados, como **java.nio.channels** y **java.nio.charset**.

El nuevo sistema de E/S está constituido por dos elementos fundamentales: *búfers* y *canales*. Un búfer contiene datos, mientras que un canal representa una conexión abierta a una dispositivo de E/S, como un archivo o una conexión. En general, para usar el nuevo sistema de E/S, usted debe obtener un canal a un dispositivo de E/S y un búfer para contener datos. Luego debe operar en el búfer y dar entrada o salida a los datos, según se requiera.

Otras dos entidades que el nuevo sistema de E/S emplea son los conjuntos de caracteres y los selectores. Un *conjunto de caracteres* define la manera en que los bytes se correlacionan con los caracteres. Usted puede codificar una secuencia de caracteres en bytes usando un *codificador* y puede decodificar una secuencia de bytes en caracteres usando un *decodificador*. Un *selector* soporta E/S de teclado, sin bloque de varias partes. En otras palabras, los selectores le permiten realizar E/S mediante varios canales. Los selectores se aplican mayormente a canales con respaldo de conexión.

Es importante comprender que el nuevo subsistema de E/S no tiene el objetivo de reemplazar a las clases de E/S que se encuentran en **java.io** y que se analizaron en este módulo. Por el contrario, las clases del nuevo sistema de E/S están diseñadas para complementar el sistema de E/S estándar, con lo que ofrecen un método alterno que puede resultar benéfico en ciertas circunstancias.

Comprobación de avance

1. ¿Qué clase se usa para leer caracteres de un archivo?
2. ¿Qué clase se usa para escribir caracteres en un archivo?

HABILIDAD FUNDAMENTAL 10.12

Uso de los envolventes de tipo de Java para convertir cadenas numéricas

Antes de abandonar el tema de la E/S, examinaremos una técnica útil al momento de leer cadenas numéricas. Como ya lo sabe, el método **println( )** de Java proporciona una manera eficiente de dar salida a varios tipos de datos a la consola, incluyendo valores numéricos de los tipos integrados, como **int** y **double**. Por lo tanto, **println( )** convierte automáticamente valores numéricos a una forma legible para el ser humano. Sin embargo, Java no proporciona un método de entrada que sea paralelo y que lea y convierta cadenas que contengan valores numéricos a su formato interno y binario. Por ejemplo, no hay una versión de **read( )** que lea una cadena como "100" y luego la convierta automáticamente a su valor binario correspondiente para que éste pueda almacenarse en una variable **int**. En cambio, Java proporciona otras maneras de realizar esta tarea. Tal vez la más fácil consista en usar uno de los *envolventes de tipo* de Java.

Los envolventes de tipo de Java son clases que encapsulan, o *envuelven*, los tipos primitivos. Los envolventes de tipo resultan necesarios porque los tipos primitivos no son objetos, hecho que limita su uso en cierta medida. Por ejemplo, un tipo primitivo no puede pasarse a una referencia. Para atender este tipo de necesidad, Java proporciona clases que corresponden a cada uno de los tipos primitivos.

Los envolventes de tipo son **Double, Float, Long, Integer, Short, Byte, Character** y **Boolean**. Estas clases ofrecen una amplia variedad de métodos que le permiten integrar plenamente los tipos primitivos en una jerarquía de objetos de Java. Como beneficio adicional, los envolventes numéricos también definen métodos que convierten una cadena numérica en su equivalente binario correspondiente. Estos métodos de conversión se muestran a continuación. Cada uno regresa un valor binario que corresponde a la cadena.

Envolvente	Método de conversión
Double	static double parseDouble(String *cad*) throws NumberFormatException
Float	static float parseFloat(String *cad*) throws NumberFormatException

1. Para leer caracteres, use **FileReader**.
2. Para escribir caracteres, use **FileWriter**.

Envolvente	Método de conversión
Long	static long parseLong(String *cad*) throws NumberFormatException
Integer	static int parseInt(String *cad*) throws NumberFormatException
Short	static short parseShort(String *cad*) throws NumberFormatException
Byte	static byte parseByte(String *cad*) throws NumberFormatException

Los envolventes de entero ofrecen también un segundo método de análisis que le permite especificar la base.

Los métodos de análisis brindan una manera fácil de convertir un valor numérico, leído como cadena del teclado o un archivo de texto, a su formato interno apropiado. Por ejemplo, el siguiente programa demuestra **parseInt()** y **parseDouble()**. El programa proporciona el promedio de una lista de números ingresados por el usuario: primero le pide al usuario el número de valores del que se obtendrá el promedio; luego lee esos números usando **readLine()** y utiliza **parseInt()** para convertir la cadena en un entero. A continuación, da entrada a los valores empleando **parseDouble()** para convertir las cadenas a sus equivalentes **double**.

```
/* Este programa da el promedio de una lista de números
   ingresados por el usuario.  */

import java.io.*;

class PromNums {
  public static void main(String args[])
    throws IOException
  {
    // crea un BufferedReader usando System.in
    BufferedReader br = new
      BufferedReader(new InputStreamReader(System.in));
    String cad;
    int n;
    double sum = 0.0;
    double prom, t;

    System.out.print("Cuántos números ingresará?: ");
    cad = br.readLine();
    try {
      n = Integer.parseInt(cad); <-- Convierte la cadena en int.
    }
    catch(NumberFormatException exc) {
      System.out.println("Formato no válido");
      n = 0;
    }

    System.out.println("Ingrese " + n + " valores.");
```

```
    for(int i=0; i < n ; i++)  {
      System.out.print(": ");
      cad = br.readLine();
      try {
        t = Double.parseDouble(cad);
      } catch(NumberFormatException exc) {
        System.out.println("Formato no válido");
        t = 0.0;
      }
      sum += t;
    }
    prom = sum / n;
    System.out.println("El promedio es " + prom);
  }
}
```

Convierte la cadena en **double**.

He aquí una ejecución de ejemplo.

```
¿Cuántos números ingresará?: 5
Ingrese 5 valores.
: 1.1
: 2.2
: 3.3
: 4.4
: 5.5
El promedio es 3.3
```

Pregunte al experto

P: **¿Qué más hacen las clases de envolvente de tipo primitivo?**

R: Los envolventes de tipo primitivo proporcionan varios métodos que ayudan a integrar los tipos primitivos en la jerarquía de objeto. Por ejemplo, varios mecanismos de almacenamiento proporcionados por la biblioteca de Java, incluidos mapas, listas y conjuntos, sólo funcionan con objetos. Por ejemplo, para almacenar una **int** en una lista ésta debe envolverse en un objeto. Además, todos los envolventes de tipo tienen un método llamado **compareTo()**, que compara el valor contenido dentro del envolvente; otro llamado **equals()**, que prueba la igualdad de dos valores; y métodos que regresan el valor del objeto en varias formas. El tema de los envolventes de tipo se retomará en el módulo 12, cuando se analice el autoencuadre.

Proyecto 10.2 Creación de un sistema de ayuda en disco

`ArchivoAyuda.java`

En el proyecto 4.1 creó una clase **Ayuda** que desplegaba información acerca de las instrucciones de control de Java. En esa implementación, la información de ayuda se almacenó dentro de la propia clase y el usuario seleccionaba la ayuda de un menú de opciones numeradas. Aunque este método era totalmente funcional, ciertamente no era la manera ideal de crear un sistema de Ayuda. Por ejemplo, para agregar o cambiar la información de ayuda, el código fuente del programa requería modificarse. Además, la selección del tema por número en lugar de hacerlo por nombre, resulta tediosa e inapropiada para listas largas de temas. Remediaremos estas desventajas mediante la creación de un sistema de ayuda en disco.

El sistema de ayuda en disco almacena información de ayuda en un archivo. Este archivo de ayuda es de texto estándar y puede cambiarse o expandirse a voluntad, sin cambiar el programa de ayuda. El usuario obtiene ayuda acerca de un tema al escribir su nombre. El sistema de ayuda busca el archivo de ayuda por tema. Una vez encontrado, la información del tema se despliega.

Paso a paso

1. Cree un archivo que sea utilizado por el sistema de ayuda. El archivo de ayuda es un archivo de texto estándar que se organiza de la siguiente manera:

```
#nombre-tema1
infotema

#nombre-tema2
infotema

.
.
.
#nombre-temaN
infotema
```

El nombre de cada tema debe estar antecedido por #, mientras que el nombre del tema debe estar contenido en una sola línea. Preceder cada nombre de tema con # permite que el programa encuentre rápidamente el inicio de cada tema. Después del nombre del tema, puede encontrar cualquier número de líneas de información acerca de éste. Sin embargo, debe haber una línea en blanco entre el final de la información del tema y el inicio del siguiente tema. Además, no debe haber espacios al final de la línea.

(continúa)

A continuación se presenta un archivo de ayuda simple que puede emplear para probar el sistema de ayuda en disco. Almacena información acerca de las instrucciones de control de Java.

```
#if
if(condición) instrucción;
else instrucción;

#switch
switch(expresión) {
  case constante:
    instrucción secuencia
    break;
    // ...
  }

#for
for(init; condición; iteración) instrucción;

#while
while(condición) instrucción;

#do
do {
  instrucción;
} while (condición);

#break
break; o break etiqueta;

#continue
continue; o continue etiqueta;
```

Llame a este archivo **AyudaArchivo.txt**.

2. Cree un archivo llamado **ArchivoAyuda.java**.

3. Empiece a crear la nueva clase **Ayuda** con estas líneas de código:

```
class Ayuda {
  String ayudaarchivo; // nombre del archivo de ayuda

  Ayuda(String nombrearch) {
    ayudaarchivo = nombrearch;
  }
```

El nombre del archivo de ayuda se pasa al constructor **Ayuda** y se almacena en la variable de instancia **archivoayuda**. Debido a que cada instancia de **Ayuda** tendrá su propia copia de **archivoayuda**, cada instancia puede usar un archivo diferente. De esa manera puede crear diferentes conjuntos de archivos de ayuda para diferentes conjuntos de temas.

4. Agregue un método **ayudaen()** (el cual se muestra a continuación) a la clase **Ayuda**. Este método recupera ayuda sobre el tema especificado.

```
// Despliega ayuda sobre un tema.
  boolean ayudaen(String que) {
    FileReader fr;
    BufferedReader ayudaLeer;
    int ch;
    String tema, info;

    try {
      fr = new FileReader(ayudaarchivo);
      ayudaLeer = new BufferedReader(fr);
    }
    catch(FileNotFoundException exc) {
      System.out.println("No se encuentra el archivo de ayuda.");
      return false;
    }
    catch(IOException exc) {
      System.out.println("No se puede abrir el archivo.");
      return false;
    }

    try {
      do {
        // lee caracteres hasta que se encuentra #
        ch = ayudaLeer.read();

         // ahora, ve si los temas coinciden
        if(ch == '#') {
          tema = ayudaLeer.readLine();
          if(que.compareTo(tema) == 0) { // tema encontrado
            do {
              info = ayudaLeer.readLine();
              if(info != null) System.out.println(info);
            } while((info != null) &&
                    (info.compareTo("") != 0));
```

(continúa)

```
          return true;
        }
      }
    } while(ch != -1);
  }
  catch(IOException exc) {
    System.out.println("Error de archivo.");
    try {
      ayudaLeer.close();
    }
    catch(IOException exc2) {
      System.out.println("Error al cerrar el archivo.");
    }
    return false;
  }
  try {
    ayudaLeer.close();
  }
  catch(IOException exc) {
    System.out.println("Error al cerrar el archivo.");
  }
  return false; // tema no encontrado
}
```

Lo primero que hay que observar es que **ayudaen()** maneja todas las posibles excepciones de E/S y ni siquiera incluye una cláusula **throws**. Al manejar sus propias excepciones, evita que esta carga se pase a todo el código que la usa. Otro código puede simplemente llamar a **ayudaen()** sin tener que envolver la llamada en un bloque **try/catch**.

El archivo de ayuda se abre usando una clase **FileReader** que está envuelta en una **BufferedReader**. Como el sistema de ayuda contiene texto, el uso de un flujo de caracteres permite que el sistema de ayuda sea internacionalizado de manera más eficiente.

El método **ayudaen()** funciona de la siguiente manera: una cadena contiene el nombre del tema que se pasa al parámetro **que**, entonces se abre el archivo de ayuda. Luego, se revisa el archivo en busca de una coincidencia entre **que** y el tema en el archivo. Recuerde que, en el archivo, cada tema se antecede con #, de modo que el bucle busca los # en el archivo. Cuando encuentra uno, revisa si el tema que sigue a ese # coincide con el que se pasó en **que**. Si es así, se despliega la información asociada con ese tema. Si se encuentra una coincidencia, **ayudaen()** regresa **true**. De otra manera, regresa **false**.

5. La clase **Ayuda** también proporciona un método llamado **obtenerSelec()**. Le pide al usuario un tema y regresa la cada de tema ingresada por el usuario.

```
// Obtiene un tema de ayuda.
  String obtenerSelec() {
    String tema = "";
```

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

    System.out.print("Ingrese tema: ");
    try {
      tema = br.readLine();
    }
    catch(IOException exc) {
      System.out.println("Error al leer la consola.");
    }
    return tema;
  }
}
```

Este método crea una clase **BufferedReader** adjunta a **System.in**. Luego pide el nombre de un tema, lee el tema y lo devuelve al que llama.

6. Aquí se muestra el sistema de ayuda en disco completo.

```
/*
   Proyecto 10.2

   Un programa de ayuda que usa un archivo
   de disco para almacenar información de ayuda.
*/

import java.io.*;

/* La clase Ayuda abre un archivo de ayuda,
   busca un tema y luego despliega la
   información asociada con ese tema.
   Observe que maneja todas las excepciones
   de E/S evitando la necesidad de llamar
   al código para hacerlo. */
class Ayuda {
  String ayudaarchivo; // nombre del archivo de ayuda

  Ayuda(String nombrearch) {
    ayudaarchivo = nombrearch;
  }

  // Despliega ayuda sobre un tema.
  boolean ayudaen(String que) {
    FileReader fr;
    BufferedReader ayudaLeer;
    int ch;
    String tema, info;
```

(continúa)

```
try {
  fr = new FileReader(ayudaarchivo);
  ayudaLeer = new BufferedReader(fr);
}
catch(FileNotFoundException exc) {
  System.out.println("No se encuentra el archivo de ayuda.");
  return false;
}
catch(IOException exc) {
  System.out.println("No se puede abrir el archivo.");
  return false;
}

try {
  do {
    // lee caracteres hasta que se encuentra #
    ch = ayudaLeer.read();

    // ahora, ve si los temas coinciden
    if(ch == '#') {
      tema = ayudaLeer.readLine();
      if(que.compareTo(tema) == 0) { // tema encontrado
        do {
          info = ayudaLeer.readLine();
          if(info != null) System.out.println(info);
        } while((info != null) &&
                (info.compareTo("") != 0));
        return true;
      }
    }
  } while(ch != -1);
}
catch(IOException exc) {
  System.out.println("Error de archivo.");
  try {
    ayudaLeer.close();
  }
  catch(IOException exc2) {
    System.out.println("Error al cerrar el archivo.");
  }
  return false;
}
try {
```

```
      ayudaLeer.close();
    }
    catch(IOException exc) {
      System.out.println("Error al cerrar el archivo.");
    }
    return false; // tema no encontrado
  }

  // Obtiene un tema de ayuda.
  String obtenerSelec() {
    String tema = "";

    BufferedReader br = new BufferedReader(
              new InputStreamReader(System.in));

    System.out.print("Ingrese tema: ");
    try {
      tema = br.readLine();
    }
    catch(IOException exc) {
      System.out.println("Error al leer la consola.");
    }
    return tema;
  }
}

// Demuestra el sistema de ayuda basado en archivo.
class ArchivoAyuda {
  public static void main(String args[]) {
    Ayuda objayuda = new Ayuda("AyudaArchivo.txt");
    String tema;

    System.out.println("Pruebe el sistema de ayuda. " +
                       "Escriba 'alto' para terminar.");
    do {
      tema = objayuda.obtenerSelec();

      if(!objayuda.ayudaen(tema))
        System.out.println("tema no encontrado.\n");

    } while(tema.compareTo("alto") != 0);
  }
}
```

Comprobación de dominio del módulo 10

1. ¿Por qué Java define flujos de bytes y caracteres?
2. Si bien la entrada y la salida de la consola son de texto, ¿por qué Java usa aún flujos de bytes para este fin?
3. Muestre cómo abrir un archivo para lectura de bytes.
4. Muestre cómo abrir un archivo para lectura de caracteres.
5. Muestre cómo abrir un archivo para E/S de acceso directo.
6. ¿Cómo puede convertir una cadena numérica como “123.23” en su equivalente binario?
7. Escriba un programa que copie un archivo de texto. En el proceso, haga que convierta todos los espacios en guiones. Use las clases de archivo de flujo de bytes.
8. Reescriba el programa descrito en la pregunta 7 para que use las clases de flujo de caracteres.
9. ¿Qué clase de flujo es **System.in**?
10. ¿Qué regresa el método **read()** de **InputStream** cuando se alcanza el final del flujo?
11. ¿Qué tipo de flujo se usa para leer datos binarios?
12. **Reader** y **Writer** constituyen la parte superior de la jerarquía de clases de_____.

Módulo 11

Programación con varios subprocesos

HABILIDADES FUNDAMENTALES

Aunque Java contiene muchas características innovadoras, una de las más excitantes es el soporte integrado a la *programación con subprocesos múltiples*. Un programa con subprocesos múltiples contiene dos o más partes que pueden ejecutarse de manera concurrente. A cada parte de este tipo de programa se le llama *subproceso* y cada subproceso define una ruta separada de ejecución. De ahí que los subprocesos múltiples sean una forma especializada de multitareas.

HABILIDAD FUNDAMENTAL 11.1

Fundamentos de los subprocesos múltiples

Hay dos tipos diferentes de subprocesos múltiples: los basados en procesos y los basados en subprocesos. Es importante comprender la diferencia entre ambos. Un proceso es, en esencia, un programa que se está ejecutando. Por consiguiente, las multitareas *basadas en procesos* son las funciones que permiten a su computadora ejecutar dos o más programas de manera concurrente. Por ejemplo una de ellas es multitareas basada en proceso, es la que le permite ejecutar el compilador de Java al mismo tiempo que está usando el editor de texto o que navega por Internet. En las multitareas basadas en procesos, un programa es la unidad más pequeña de código que puede ser despachada por el temporizador.

En el entorno de multitareas *basadas en subprocesos*, el subproceso es la unidad más pequeña de código despachable. Esto significa que un solo programa puede realizar dos o más tareas al mismo tiempo. Por ejemplo, un editor de texto puede estar formateando texto al mismo tiempo que está imprimiendo, siempre y cuando dos subprocesos separados realicen estas dos acciones. Aunque los programas de Java usan los entornos de multitareas basadas en procesos, éstos, a diferencia de las multitareas de subprocesos múltiples, no se encuentran bajo el control de Java .

La principal ventaja de los subprocesos múltiples es que permiten escribir programas muy eficientes porque posibilitan el uso del tiempo muerto que está presente en casi todos los programas. Como tal vez ya lo sabe, la mayor parte de los dispositivos de E/S, sean puertos de red, unidades de disco o teclado, son mucho más lentos que la CPU. Por lo tanto, un programa a menudo gastará la mayor parte de su tiempo de ejecución esperando enviar o recibir información de un dispositivo. Mediante el uso de los subprocesos múltiples, su programa puede ejecutar otra tarea durante el tiempo muerto. Por ejemplo, mientras una parte de su programa está enviando un archivo en Internet, tal vez otra parte lea la entrada del teclado y otra más almacene en el búfer el siguiente bloque de datos que habrá de enviarse.

Un subproceso puede estar en uno o varios estados: puede estar *ejecutándose* o puede estar listo para *ejecutarse* en cuanto obtenga tiempo de CPU. Un subproceso en ejecución puede estar *suspendido*, lo cual es una detención temporal en su ejecución y puede *reanudarse* más adelante. Un subproceso puede *bloquearse* cuando espera un recurso o puede *terminarse*, en cuyo caso su ejecución se acaba y no se reanuda.

Junto con las multitareas basadas en subprocesos se encuentra la necesidad de un tipo especial de función llamada *sincronización*, que permite que la ejecución de subprocesos esté coordinada de maneras bien definidas. Java cuenta con un subsistema completo dedicado a la sincronización, cuyas características clave también se describen aquí.

Si ha programado para sistemas operativos como Windows, entonces ya está familiarizado con la programación de subprocesos. Sin embargo, el hecho de que Java maneje subprocesos mediante elementos de lenguaje hace que los subprocesos múltiples resulten especialmente eficientes pues usted maneja un gran número de detalles.

HABILIDAD FUNDAMENTAL 11.2

La clase Thread y la interfaz Runnable

El sistema de subprocesos múltiples de Java está construido a partir de la clase **Thread** y la interfaz que la acompaña, **Runnable**. **Thread** encapsula un subproceso de ejecución. Para crear un nuevo subproceso, su programa extenderá **Thread** o implementará la interfaz **Runnable**.

La clase **Thread** define varios métodos que le permiten manejar subprocesos. He aquí algunos de los más empleados (los veremos más de cerca a medida que se utilicen).

Método	Significado
final String getName()	Obtiene el nombre de un subproceso.
final int getPriority()	Obtiene la prioridad de un subproceso.
final boolean isAlive()	Determina si un subproceso aún se está ejecutando.
final void join()	Espera a que termine un subproceso.
void run()	Punto de entrada del subproceso.
static void sleep(long *milisegundos*)	Suspende un subproceso por un periodo específico de milisegundos.
void start()	Inicia un subproceso al llamar a su método **run()**.

Todos los procesos cuentan con por lo menos un subproceso de ejecución, que se conoce por general como el *subproceso principal*, debido a que es el que se ejecuta cuando su programa inicia. Por lo tanto, el subproceso principal es el **main()** que se ha empleado en todos los programas de ejemplo de este libro. A partir del subproceso principal, pueden crearse otros subprocesos.

Comprobación de avance

1. ¿Cuál es la diferencia entre multitareas basadas en procesos y las basadas en subprocesos?
2. ¿En qué estados puede existir un subproceso?
3. ¿Cuáles clases encapsulan un subproceso?

1. Las multitareas basadas en procesos se usan para ejecutar dos o más programas de manera concurrente. Las multitareas que están basadas en subprocesos, llamadas *subprocesos múltiples*, se usan para ejecutar partes de un programa de manera concurrente.
2. Los estados de los subprocesos son *en ejecución*, *listo para ejecutarse*, *suspendido*, *bloqueado* y *terminado*. Cuando un subproceso suspendido se reinicia, se dice que se *reanuda*.
3. **Thread**.

HABILIDAD FUNDAMENTAL 11.3

Creación de un subproceso

Usted puede crear un subproceso al crear una instancia de un objeto de tipo **Thread**. La clase **Thread** encapsula un objeto que es ejecutable. Como se mencionó, Java define dos maneras en que puede crear un objeto ejecutable:

- Puede implementar la interfaz **Runnable**.
- Puede extender la clase **Thread**.

Casi todos los ejemplos de este módulo usarán el método que implementa **Runnable**. Sin embargo, en el proyecto 11.1 se muestra cómo implementar un subproceso extendiendo **Thread**. Recuerde que ambos métodos usan la clase **Thread** para la creación de la instancia, el acceso y el control del subproceso. La única diferencia es la manera en la que una clase habilitada por un subproceso se crea.

La interfaz **Runnable** abstrae una unidad de código ejecutable. Puede construir un subproceso en cualquier objeto que implemente la interfaz **Runnable**. **Runnable** sólo define un método llamado **run()**, el cual se declara de la siguiente manera:

```
public void run()
```

Se debe definir dentro de **run()** el código que constituye el nuevo subproceso. Es importante comprender que **run()** puede llamar a otros métodos, usar otras clases y declarar variables igual que el subproceso principal. La única diferencia es que **run()** establece el punto de entrada para otro subproceso, que se ejecuta de manera concurrente dentro de su programa. Este subproceso terminará cuando **run()** regrese.

Después de que haya creado una clase que implemente **Runnable**, creará una instancia de un objeto de tipo **Thread** en un objeto de esa clase. **Thread** define varios constructores. El siguiente es uno de los que usaremos primero:

Thread(Runnable *obSubproceso*)

En este constructor, *obSubproceso* es una instancia de una clase que implementa la interfaz **Runnable**. Esto define dónde empezará la ejecución del subproceso.

Una vez creado, el nuevo subproceso no empezará a ejecutarse sino hasta que llame a su método **start()**, que se declara dentro de **Thread**. En esencia, **start()** ejecuta una llamada a **run()**. El método **start()** se muestra aquí:

void start( )

He aquí un ejemplo que crea un nuevo subproceso y empieza a ejecutarlo:

```
// Crea un subproceso al implementar Runnable.
class MiSubproceso implements Runnable {
```

Objetos de **MiSubproceso** pueden ejecutarse en sus propios subprocesos porque **MiSubproceso** implementa **Runnable**.

```
  int cuenta;
  String tercerNombre;

  MiSubproceso(String nombre) {
    cuenta = 0;
    tercerNombre = nombre;
  }

  // Punto de entrada del subproceso.
  public void run() {
    System.out.println(tercerNombre + " iniciando.");
    try {
      do {
        Thread.sleep(500);
        System.out.println("En " + tercerNombre +
                           ", la cuenta es " + cuenta);
        cuenta++;
      } while(cuenta < 10);
    }
    catch(InterruptedException exc) {
      System.out.println(tercerNombre + " interrumpido.");
    }
    System.out.println(tercerNombre + " terminando.");
  }
}

class UsoSubprocesos {
  public static void main(String args[]) {
    System.out.println("Iniciando subproceso principal.");

    // Primero, construye un objeto de MiSubproceso.
    MiSubproceso ms = new MiSubproceso("descendiente #1");

    // Ahora, construye un subproceso a partir de ese objeto.
    Thread nuevoSubpr = new Thread(ms);

    // Por último, inicia la ejecución del subproceso.
    nuevoSubpr.start();

    do {
      System.out.print(".");
      try {
        Thread.sleep(100);
      }
      catch(InterruptedException exc) {
        System.out.println("Subproceso principal interrumpido.");
```

Aquí empiezan a ejecutarse los subprocesos.

Crea un objeto ejecutable.

Construye un subproceso en ese objeto.

Empieza a ejecutar el subproceso.

```
      }
    } while (ms.cuenta != 10);

    System.out.println("Subproceso principal terminando.");
  }
}
```

Revisemos de cerca este programa. En primer lugar, **MiSubproceso** implementa **Runnable**, lo cual significa que un objeto de tipo **MiSubproceso** resulta adecuado para usarlo como subproceso y puede pasarse al constructor **Thread**.

Dentro de **run()**, se establece un bucle que cuenta de 0 a 9. Observe la llamada a **sleep()**: el método **sleep()** hace que el subproceso a partir del cual es llamado suspenda la ejecución por un periodo específico de milisegundos. Ésta es su forma general:

static void sleep(long *milisegundos*) throws InterruptedException

El número de milisegundos que se suspenderá se especifica en *milisegundos*. Este método puede lanzar una **InterruptedException**. Por lo tanto, las llamadas hacia él deben envolverse en un bloque **try/catch**. El método **sleep()** también tiene una segunda forma que le permite especificar el periodo en milisegundos y nanosegundos en caso de que requiera ese nivel de precisión.

Dentro de **main()**, se crea un nuevo objeto de **Thread** mediante la siguiente secuencia de instrucciones:

```
// Primero, construye un objeto de MiSubproceso.
MiSubproceso ms = new MiSubproceso("descendiente #1");

// Ahora, construye un subproceso a partir de ese objeto.
Thread nuevoSubpr = new Thread(ms);

// Por último, inicia la ejecución del subproceso.
nuevoSubpr.start();
```

Como el comentario lo sugiere, primero se crea un objeto de **MiSubproceso**. Este objeto se usa a continuación para construir un objeto de **Thread**. Esto es posible porque **MiSubproceso** implementa **Runnable**. Por último, se inicia la ejecución de un nuevo subproceso al llamar a **start()**. Esto hace que se inicie el método **run()** del subproceso descendiente. Después de llamar a **start()**, la ejecución regresa a **main()** e ingresa en el bucle **do** de éste. Ambos subprocesos continúan ejecutándose, compartiendo la CPU, hasta que sus bucles terminan. La salida producida por este programa se mostrará enseguida. Debido a las diferencias entre los entornos de computación, la salida precisa que vea puede diferir de la que se muestra a continuación.

```
Iniciando subproceso principal.
.descendiente #1 iniciando.
....En descendiente #1, la cuenta es 0
.....En descendiente #1, la cuenta es 1
....En descendiente #1, la cuenta es 2
```

```
.....En descendiente #1, la cuenta es 3
   ....En descendiente #1, la cuenta es 4
.....En descendiente #1, la cuenta es 5
.....En descendiente #1, la cuenta es 6
   ....En descendiente #1, la cuenta es 7
.....En descendiente #1, la cuenta es 8
   ....En descendiente #1, la cuenta es 9
descendiente #1 terminando.
Subproceso principal terminando.
```

En un programa de subprocesos múltiples, a menudo querrá que el subproceso principal sea el último subproceso en terminar de ejecutarse. Técnicamente, un programa sigue ejecutándose hasta que todos los subprocesos han terminado. Así, no es obligatorio hacer que el subproceso principal termine al final. Sin embargo, es una buena práctica (especialmente cuando está aprendiendo acerca de los programas con subprocesos múltiples). En el programa anterior se asegura que el subproceso principal terminará al final, porque el bucle **do** se detiene cuando cuenta es igual a 10. Debido a que cuenta sólo será igual a 10 después de que **nuevoSubpr** haya terminado, el subproceso principal termina al final. Más adelante en este módulo verá una mejor manera de hacer que un subproceso espere hasta que otro termine.

Algunas mejoras simples

Aunque el programa anterior es perfectamente válido, pueden realizarse algunas mejoras simples que lo volverán más eficiente y fácil de usar. En primer lugar, es posible hacer que un subproceso empiece la ejecución en cuanto se cree. En el caso de **MiSubproceso**, esto se lleva a cabo al crear una instancia de un objeto **Thread** dentro del constructor de **MiSubproceso**. En segundo lugar, no hay necesidad de que **MiSubproceso** almacene el nombre del subproceso pues es posible asignar un nombre a un subproceso cuando éste se cree. Para ello, debe usar esta versión del constructor de **Thread**:

Thread(Runnable *obSubproceso*, String *nombre*)

Aquí, *nombre* se vuelve el nombre del subproceso.

Pregunte al experto

P: **¿Por qué recomienda que el subproceso principal sea el último en terminar?**

R: En antiguos sistemas de Java en tiempo de ejecución, si el subproceso principal terminaba antes que un subproceso descendiente se hubiera completado, existía la posibilidad de que el sistema en tiempo de ejecución de Java se "colgara". Este problema no aparece en los sistemas modernos en tiempo de ejecución de Java a los que este autor tiene acceso. Sin embargo, como este comportamiento era exhibido en algunos sistemas en tiempo de ejecución de Java, aparentemente es mejor estar seguro que arrepentirse, pues no siempre es posible que sepa en qué entorno se ejecutará su programa. Además, como verá, es muy fácil hacer que el subproceso principal espere hasta que los subprocesos descendientes se completen.

Para obtener el nombre de un subproceso, se llama a **getName()**, el cual está definido por **Thread**. Ésta es su forma general:

final String getName( )

Aunque no es necesario para el siguiente programa, puede establecer el nombre de un subproceso después de que se haya creado si usa **setName()**, el cual se muestra aquí:

final void setName(String *nombreSubproceso*)

Aquí, *nombreSubproceso* especifica el nombre del subproceso.

He aquí la versión mejorada del ejemplo anterior:

```
// MiSubproceso mejorado.

class MiSubproceso implements Runnable {
  int cuenta;
  Thread subpr; <-------- Una referencia al subproceso está almacenada en subpr.

  // Construye un nuevo subproceso.
  MiSubproceso(String nombre) {
    subpr = new Thread(this, nombre); <-- Se asigna un nombre al
    cuenta = 0;                           subproceso cuando éste se crea.
    subpr.start(); // inicia el subproceso
  }

  // Empieza la ejecución de un nuevo subproceso.
  public void run() {
    System.out.println(subpr.getName() + " iniciando.");
    try {
      do {
        Thread.sleep(500);
        System.out.println("En " + subpr.getName() +
                           ", La cuenta es " + cuenta);
        cuenta++;
      } while(cuenta < 10);
    }
    catch(InterruptedException exc) {
      System.out.println(subpr.getName() + " interrumpido.");
    }
    System.out.println(subpr.getName() + " terminando.");
  }
}

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

```
    System.out.println("Subproceso principal iniciando.");

    MiSubproceso ms = new MiSubproceso("descendiente #1");
                                                          Ahora el subproceso inicia cuando se crea.
    do {
      System.out.print(".");
      try {
        Thread.sleep(100);
      }
      catch(InterruptedException exc) {
        System.out.println("Subproceso principal interrumpido.");
      }
    } while (ms.cuenta != 10);

    System.out.println("Subproceso principal terminando.");
  }
}
```

Esta versión produce la misma salida que se mostró antes. Observe que el subproceso está almacenado en **subpr** dentro de **MiSubproceso**.

Comprobación de avance

1. ¿Cuáles son las dos maneras en las que se crea una clase que puede actuar como un subproceso?
2. ¿Cuál es el propósito del método **run()** definido por **Runnable**?
3. ¿Qué hace el método **start()** definido por **Thread**?

Proyecto 11.1 Extensión de Thread

`ExtiendeThread.java`

La implementación de **Runnable** representa una manera de crear una clase que puede crear instancias de objetos de subprocesos. La extensión de **Thread** es otra. En este proyecto, verá cómo extender **Thread** al crear un programa funcionalmente idéntico al programa **UsoSubprocesoMejorado**.

(continúa)

1. Para crear un subproceso, debe implementar **Runnable** o extender **Thread**.
2. El método **run()** es el punto de entrada a un subproceso.
3. El método **start()** inicia la ejecución de un subproceso.

Cuando una clase extiende **Thread**, debe sobrescribir el método **run()**, que es el punto de entrada para el nuevo subproceso. También debe llamar a **start()** con el fin de iniciar la ejecución del nuevo subproceso. Es posible sobrescribir otros métodos **Thread**, pero no es obligatorio hacerlo.

Paso a paso

1. Cree un archivo llamado **ExtiendeThread.java**. En este archivo, copie el código del ejemplo de subprocesos (**UsoSubprocesoMejorado.java**).
2. Cambie la declaración **MiSubproceso** para que extienda **Thread** en lugar de implementar **Runnable**, como se muestra aquí.

```
class MiSubproceso extends Thread {
```

3. Elimine esta línea:

```
Thead subpr
```

La variable **subpr** ya no es necesaria porque **MiSubproceso** incluye una instancia de **Thread** y puede hacer referencia a sí misma.

4. Cambie el constructor de **MiSubproceso** para que tenga este aspecto:

```
// Construye un nuevo subproceso.
MiSubproceso(String nombre) {
  super(nombre);  // nombre del subproceso
  cuenta = 0;
  start(); // inicia el subproceso
}
```

Como verá, primero se usa súper para llamar a esta versión del constructor de **Thread**:

Thread(String *nombre*)

Aquí, nombre es el nombre del subproceso. El objeto que se ejecutará es el subproceso que invoca, que en este caso es el subproceso que se está creando.

5. Cambie **run()** para que llame directamente a **getName()**, sin calificarlo con la variable **subpr**. Deberá tener el siguiente aspecto.

```
// Empieza la ejecución de un nuevo subproceso.
public void run() {
  System.out.println(getName() + " iniciando.");
  try {
    do {
      Thread.sleep(500);
      System.out.println("En " + getName() +
                         ", La cuenta es " + cuenta);
```

```
      cuenta++;
    } while(cuenta < 10);
  }
  catch(InterruptedException exc) {
    System.out.println(getName() + " interrumpido.");
  }
  System.out.println(getName() + " terminando.");
}
```

6. He aquí el programa completo, que ahora extiende **Thread** en lugar de implementar **Runnable**. La salida es la misma que la de antes.

```
/*
   Proyecto 11.1

   Extiende Thread.
*/

class MiSubproceso extends Thread {
  int cuenta;

  // Construye un nuevo subproceso.
  MiSubproceso(String nombre) {
    súper(nombre);  // nombre del subproceso
    cuenta = 0;
    start(); // inicia el subproceso
  }

  // Empieza la ejecución de un nuevo subproceso.
  public void run() {
    System.out.println(getName() + " iniciando.");
    try {
      do {
        Thread.sleep(500);
        System.out.println("En " + getName() +
                           ", La cuenta es " + cuenta);
        cuenta++;
      } while(cuenta < 10);
    }
    catch(InterruptedException exc) {
      System.out.println(getName() + " interrumpido.");
    }
    System.out.println(getName() + " terminando.");
  }
}

class UsoSubprocesosMejorado {
```

(continúa)

```
  public static void main(String args[]) {
    System.out.println("Subproceso principal iniciando.");

    MiSubproceso ms = new MiSubproceso("descendiente #1");

    do {
      System.out.print(".");
      try {
        Thread.sleep(100);
      }
      catch(InterruptedException exc) {
        System.out.println("Subproceso principal interrumpido.");
      }
    } while (ms.cuenta != 10);

    System.out.println("Subproceso principal terminando.");
  }
}
```

HABILIDAD FUNDAMENTAL 11.4

Creación de subprocesos múltiples

En los ejemplos anteriores sólo se ha creado un subproceso descendiente. Sin embargo, su programa puede abarcar todos los subprocesos que sean necesarios. Por ejemplo, el siguiente programa crea tres subprocesos descendientes:

```
// Crea subprocesos múltiples.

class MiSubproceso implements Runnable {
  int cuenta;
  Thread subpr;

  // Construye un nuevo subproceso.
  MiSubproceso(String nombre) {
    subpr = new Thread(this, nombre);
    cuenta = 0;
    subpr.start(); // inicia el subproceso
  }

  // Inicia la ejecución del nuevo subproceso.
  public void run() {
    System.out.println(subpr.getName() + " iniciando.");
    try {
      do {
        Thread.sleep(500);
        System.out.println("En " + subpr.getName() +
                           ", La cuenta es " + cuenta);
```

```
        cuenta++;
      } while(cuenta < 10);
    }
    catch(InterruptedException exc) {
      System.out.println(subpr.getName() + " interrumpido.");
    }
    System.out.println(subpr.getName() + " terminando.");
  }
}

class SubprocesosAdicionales {
  public static void main(String args[]) {
    System.out.println("Subproceso principal iniciando.");

    MiSubproceso ms1 = new MiSubproceso("descendiente #1");
    MiSubproceso ms2 = new MiSubproceso("descendiente #2");
    MiSubproceso ms3 = new MiSubproceso("descendiente #3");

    do {
      System.out.print(".");
      try {
        Thread.sleep(100);
      }
      catch(InterruptedException exc) {
        System.out.println("Subproceso principal interrumpido.");
      }
    } while (ms1.cuenta < 10 ||
             ms2.cuenta < 10 ||
             ms3.cuenta < 10);

    System.out.println("Subproceso principal terminando.");
  }
}
```

Crea e inicia la ejecución de tres subprocesos.

Pregunte al experto

P: **¿Por qué Java tiene dos maneras de crear subprocesos descendientes (extender** Thread **o implementar** Runnable**) y cuál método es mejor?**

R: La clase **Thread** define varios métodos que una clase derivada puede sobrescribir. De estos métodos, el único que debe sobrescribirse es **run( )**. Se trata, por supuesto, del mismo método que se requiere cuando implementa **Runnable**. Algunos programadores de Java sienten que esas clases sólo deben extenderse cuando se mejoran o modifican de alguna manera. Así, si no sobrescribe ninguno de los otros métodos de **Thread**, probablemente lo mejor sea solamente implementar **Runnable**. Esto, por supuesto, depende de usted.

A continuación se presenta una salida de ejemplo de este programa. (La salida que vea puede ser ligeramente diferente.)

```
Subproceso principal iniciando.
.descendiente #1 iniciando.
descendiente #2 iniciando.
descendiente #3 iniciando.
.....En descendiente #1, La cuenta es 0
En descendiente #2, La cuenta es 0
En descendiente #3, La cuenta es 0
.....En descendiente #1, La cuenta es 1
En descendiente #2, La cuenta es 1
En descendiente #3, La cuenta es 1
.....En descendiente #1, La cuenta es 2
En descendiente #2, La cuenta es 2
En descendiente #3, La cuenta es 2
.....En descendiente #1, La cuenta es 3
En descendiente #2, La cuenta es 3
En descendiente #3, La cuenta es 3
.....En descendiente #1, La cuenta es 4
En descendiente #2, La cuenta es 4
En descendiente #3, La cuenta es 4
.....En descendiente #1, La cuenta es 5
En descendiente #2, La cuenta es 5
En descendiente #3, La cuenta es 5
.....En descendiente #1, La cuenta es 6
En descendiente #2, La cuenta es 6
En descendiente #3, La cuenta es 6
.....En descendiente #1, La cuenta es 7
En descendiente #2, La cuenta es 7
En descendiente #3, La cuenta es 7
.....En descendiente #1, La cuenta es 8
En descendiente #2, La cuenta es 8
En descendiente #3, La cuenta es 8
.....En descendiente #1, La cuenta es 9
descendiente #1 terminando.
En descendiente #2, La cuenta es 9
descendiente #2 terminando.
En descendiente #3, La cuenta es 9
descendiente #3 terminando.
Subproceso principal terminando.
```

Como puede ver, una vez iniciados, los tres subprocesos descendientes comparten la CPU. Observe que los subprocesos inician en el orden en que se crearon. Sin embargo, tal vez éste no sea siempre el caso. Java tiene la libertad de programar la ejecución de los subprocesos a su propia manera. Por supuesto, debido a las diferencias en el cronómetro o el entorno, la salida precisa del programa puede ser diferente, de modo que no se sorprenda si ve resultados ligeramente diferentes cuando pruebe el programa.

HABILIDAD FUNDAMENTAL 11.5

Cómo determinar cuándo termina un subproceso

A menudo es útil saber cuándo ha terminado un subproceso. En los ejemplos anteriores, logramos esto al observar la variable **cuenta**, pero esto apenas es una solución satisfactoria o generalizable. Por fortuna, **Thread** proporciona dos medios mediante los cuales puede determinar si un subproceso ha terminado. En primer lugar, puede llamar a **isAlive()** en el subproceso. Ésta es su forma general:

final boolean isAlive()

El método **isAlive()** regresa **true** si el subproceso en el que es llamado se está ejecutando todavía. De otra manera, regresa **false**. Para probar **isAlive()**, sustituya esta versión de **SubprocesosAdicionales** por la que se mostró en el programa anterior.

```
// Uso de isAlive().
class SubprocesosAdicionales {
  public static void main(String args[]) {
    System.out.println("Subproceso principal iniciando.");

    MiSubproceso ms1 = new MiSubproceso("descendiente #1");
    MiSubproceso ms2 = new MiSubproceso("descendiente #2");
    MiSubproceso ms3 = new MiSubproceso("descendiente #3");

    do {
      System.out.print(".");
      try {
        Thread.sleep(100);
      }
      catch(InterruptedException exc) {
        System.out.println("Subproceso principal interrumpido.");
      }
    } while (ms1.subpr.isAlive() ||
             ms2.subpr.isAlive() ||  <-- Esto espera hasta que terminan todos los subprocesos.
             ms3.subpr.isAlive());

    System.out.println("Subproceso principal terminando.");
  }
}
```

Esta versión produce la misma salida que la de antes. La única diferencia es que usa **isAlive()** para esperar a que los subprocesos descendientes terminen.

Otra manera de esperar a que un subproceso termine es llamar a **join()**, el cual se muestra aquí:

final void join( ) throws InterruptedException

Este método espera hasta que el subproceso en el que es llamado termine. Su nombre proviene del concepto de que la llamada al subproceso espera hasta que el subproceso especificado se les *une*. Formas adicionales de **join( )** le permiten especificar la cantidad máxima de tiempo que desee esperar a que el subproceso especificado termine.

He aquí un programa que usa **join( )** para asegurar que el subproceso principal sea el último en detenerse.

```
// Uso de join().

class MiSubproceso implements Runnable {
  int cuenta;
  Thread subpr;

  // Construye un nuevo subproceso.
  MiSubproceso(String nombre) {
    subpr = new Thread(this, nombre);
    cuenta = 0;
    subpr.start(); // inicia el subproceso
  }

  // Empieza la ejecución del nuevo subproceso.
  public void run() {
    System.out.println(subpr.getName() + " iniciando.");
    try {
      do {
        Thread.sleep(500);
        System.out.println("En " + subpr.getName() +
                           ", La cuenta es " + cuenta);
        cuenta++;
      } while(cuenta < 10);
    }
    catch(InterruptedException exc) {
      System.out.println(subpr.getName() + " interrumpido.");
    }
    System.out.println(subpr.getName() + " terminando.");
  }
}

class SubprocesosUnidos {
  public static void main(String args[]) {
    System.out.println("Subproceso principal iniciando.");

    MiSubproceso ms1 = new MiSubproceso("descendiente #1");
    MiSubproceso ms2 = new MiSubproceso("descendiente #2");
    MiSubproceso ms3 = new MiSubproceso("descendiente #3");
```

```
    try {
      ms1.subpr.join();
      System.out.println("descendiente #1 unido.");
      ms2.subpr.join();
      System.out.println("descendiente #2 unido.");
      ms3.subpr.join();
      System.out.println("descendiente #3 unido.");
    }
    catch(InterruptedException exc) {
      System.out.println("Subproceso principal  interrumpido.");
    }

    System.out.println("Subproceso principal terminando.");
  }
}
```

Espera hasta que termina el subproceso especificado.

A continuación se muestra la salida de este programa. Recuerde que cuando pruebe el programa, su salida precisa puede llegar a variar ligeramente.

```
Subproceso principal iniciando.
descendiente #1 iniciando.
descendiente #2 iniciando.
descendiente #3 iniciando.
En descendiente #1, La cuenta es 0
En descendiente #2, La cuenta es 0
En descendiente #3, La cuenta es 0
En descendiente #1, La cuenta es 1
En descendiente #2, La cuenta es 1
En descendiente #3, La cuenta es 1
En descendiente #1, La cuenta es 2
En descendiente #2, La cuenta es 2
En descendiente #3, La cuenta es 2
En descendiente #1, La cuenta es 3
En descendiente #2, La cuenta es 3
En descendiente #3, La cuenta es 3
En descendiente #1, La cuenta es 4
En descendiente #2, La cuenta es 4
En descendiente #3, La cuenta es 4
En descendiente #1, La cuenta es 5
En descendiente #2, La cuenta es 5
En descendiente #3, La cuenta es 5
En descendiente #1, La cuenta es 6
En descendiente #2, La cuenta es 6
En descendiente #3, La cuenta es 6
En descendiente #1, La cuenta es 7
En descendiente #2, La cuenta es 7
```

```
En descendiente #3, La cuenta es 7
En descendiente #3, La cuenta es 8
En descendiente #2, La cuenta es 8
En descendiente #1, La cuenta es 8
En descendiente #3, La cuenta es 9
descendiente #3 terminando.
En descendiente #2, La cuenta es 9
descendiente #2 terminando.
En descendiente #1, La cuenta es 9
descendiente #1 terminando.
descendiente #1 unido.
descendiente #2 unido.
descendiente #3 unido.
Subproceso principal terminando.
```

Como puede ver, después de que las llamadas a **join( )** regresan, los subprocesos dejan de ejecutarse.

Comprobación de avance

1. ¿Cuáles son las dos maneras en las que puede determinar si ha terminado un subproceso?
2. Explique en qué consiste **join( )**.

HABILIDAD FUNDAMENTAL 11.6

Prioridades en subprocesos

Cada subproceso tiene un parámetro asociado de prioridad. La prioridad de un subproceso determina, en parte, cuánto tiempo de la CPU recibe un subproceso. En general, los subprocesos de baja prioridad reciben poco, en tanto que los de alta prioridad reciben demasiado. Como es de esperarse, la cantidad de tiempo de CPU que un subproceso recibe tiene un impacto profundo en sus características de ejecución y su interacción con otros subprocesos que se ejecutan en el sistema.

Es importante comprender que otros factores, además de la prioridad del subproceso, afectan también a la cantidad de tiempo de CPU que un subproceso recibe. Por ejemplo, si un subproceso de alta prioridad está esperando algún recurso, tal vez la entrada del teclado, entonces se bloqueará y se ejecutará un subproceso de baja prioridad. Sin embargo, cuando el subproceso de alta prioridad obtiene acceso al recurso, puede hacer a un lado al subproceso de baja prioridad y reanudar la

1. Para determinar si un subproceso ha terminado, puede llamar a **isAlive( )** o usar **join( )** para esperar a que el subproceso se una al subproceso que llama.
2. El método **join( )** suspende la ejecución del subproceso que llama hasta que el subproceso en el que es llamado termine.

ejecución. Otro factor que afecta la programación de los subprocesos es la manera en la que el sistema operativo implementa las multitareas. (Consulte "Pregunte al experto" al final de esta sección.) Por lo tanto, el hecho de darle a un subproceso una alta prioridad y a otro una baja no necesariamente significa que un subproceso se ejecutará más rápido o con más frecuencia que otro. Tan sólo sucede que el subproceso de alta prioridad tiene mayores posibilidades de acceso al CPU.

Cuando un subproceso secundario se inicia, su parámetro de prioridad es igual al de su subproceso progenitor. Puede cambiar la prioridad de un subproceso si llama a **setPriority()**, el cual es un miembro de **Thread**. He aquí su forma general:

final void setPriority(int *nivel*).

Aquí, *nivel* especifica el nuevo parámetro de prioridad para el subproceso que llama. El valor de *nivel* debe estar dentro del rango **MIN_PRIORITY** y **MAX_PRIORITY**. Actualmente, estos valores son 1 y 10, respectivamente. Para regresar un subproceso a su prioridad predeterminada, especifique **NORM_PRIORITY**, que actualmente es 5. Estas prioridades se definen como variables final dentro de **Thread**.

Puede obtener el valor de prioridad actual llamando al método **getPriority()** de **Thread** que se muestra aquí:

final int getPriority()

El siguiente ejemplo demuestra dos subprocesos con diferentes prioridades. Los subprocesos se crean como instancias de **Priority**. El método **run()** contiene un bucle que cuenta el número de iteraciones. El bucle se detiene cuando la cuenta alcanza 10 000 000 o la variable estática alto es **true**. Inicialmente, **alto** se establece como **false**, pero el primer subproceso en terminar su conteo asigna **true** a **alto**. Esto hace que el segundo subproceso termine con su siguiente porción de tiempo. Cada vez que el bucle se recorre, la cadena en **nombreActual** se compara con el nombre del subproceso en ejecución. Si no coincide, significa entonces que un interruptor de tarea se ha presentado. Cada vez que esto se presenta, se despliega el nombre del nuevo subproceso, y a **nombreActual** se le asigna el nombre del nuevo subproceso. Esto le permite observar la frecuencia con que cada subproceso tiene acceso al CPU. Después de que ambos subprocesos se detienen, se despliega el número de iteraciones para cada bucle.

```
// Demuestra las prioridades de los subprocesos.

class Prioridad implements Runnable {
  int cuenta;
  Thread subpr;

  static boolean alto = false;
  static String nombreActual;

  /* Construye un nuevo subproceso. Observe que este
     constructor no inicia realmente la
     ejecución de los subprocesos. */
```

```
  Prioridad(String nombre) {
    subpr = new Thread(this, nombre);
    cuenta = 0;
    nombreActual = nombre;
  }

  // Empieza la ejecución del nuevo subproceso.
  public void run() {
    System.out.println(subpr.getName() + " iniciando.");
    do {
      cuenta++;

      if(nombreActual.compareTo(subpr.getName()) != 0) {
        nombreActual = subpr.getName();
        System.out.println("En " + nombreActual);
      }

    } while(alto == false && cuenta < 10000000);
    alto = true;

    System.out.println("\n" + subpr.getName() +
                       " terminando.");
  }
}

class PrioridadDemo {
  public static void main(String args[]) {
    Prioridad ms1 = new Prioridad("Alta prioridad");
    Prioridad ms2 = new Prioritdad("Baja prioridad");

    // establece las prioridades
    ms1.subpr.setPriority(Thread.NORM_PRIORITY+2);
    ms2.subpr.setPriority(Thread.NORM_PRIORITY-2);

    // inicia los subprocesos
    ms1.subpr.start();
    ms2.subpr.start();

    try {
      ms1.subpr.join();
      ms2.subpr.join();
    }
    catch(InterruptedException exc) {
      System.out.println("Subproceso principal interrumpido.");
    }
```

El primer subproceso en 10000000 detiene todos los subprocesos.

Le da a **ms1** una mayor prioridad que a **ms2**.

```
    System.out.println("\nCuenta del subproceso de alta prioridad " +
                       ms1.cuenta);
    System.out.println("Cuenta del subproceso de baja prioridad " +
                       ms2.cuenta);
  }
}
```

He aquí una ejecución de ejemplo:

```
Alta prioridad iniciando.
En Alta prioridad
Baja prioridad iniciando.
En Baja prioridad
En Alta prioridad

Alta prioridad terminando.

Baja prioridad terminando.

Cuenta del subproceso de alta prioridad 10000000
Cuenta del subproceso de baja prioridad 171609
```

En esta ejecución, el subproceso de alta prioridad obtiene la mayor parte del tiempo de la CPU. Por supuesto, la salida exacta producida por este programa dependerá de la velocidad de su CPU, el sistema operativo que esté empleando y la cantidad de tareas adicionales que esté ejecutando en su sistema.

Pregunte al experto

P: **¿Qué tanto afecta la implementación que hace el sistema operativo de las multitareas a la cantidad de tiempo de CPU que un subproceso recibe?**

R: Además del valor de prioridad del subproceso, el factor más importante que afecta la ejecución es la manera en que el sistema operativo implementa las multitareas y la programación. Algunos sistemas operativos usan multitareas con derecho de uso en el que cada subproceso recibe una parte del tiempo, por lo menos ocasionalmente. Otros sistemas usan programación sin derecho en el que un subproceso debe ejecutarse antes de que otro subproceso lo ejecute. En los sistemas sin derechos, es fácil para un subproceso dominar, evitando que otros se ejecuten.

HABILIDAD FUNDAMENTAL 11.7

Sincronización

Cuando se usan subprocesos múltiples, a veces es necesario coordinar las actividades de dos o más de ellos. El proceso con que se logra esto es la *sincronización*. La razón más común para la sincronización es cuando dos o más subprocesos necesitan acceder a un recurso compartido que sólo puede ser usado por un subproceso al mismo tiempo. Por ejemplo, cuando un subproceso está escribiendo en un archivo, debe evitarse que un segundo subproceso haga lo mismo al mismo tiempo. Otra razón para la sincronización es cuando un subproceso está esperando un evento provocado por otro. En este caso, debe haber algunos medios mediante los cuales el primer subproceso pueda mantenerse en estado suspendido hasta que el evento haya ocurrido. Así, el subproceso que espera debe reanudar la ejecución.

La clave para la sincronización en Java es el concepto de *monitor*, que controla el acceso a un objeto. Un monitor funciona al implementar el concepto de *bloqueo*. Cuando un objeto está bloqueado por un subproceso, ningún subproceso puede tener acceso a dicho objeto. Cuando existe el subproceso, el objeto está desbloqueado y se encuentra disponible para que otro subproceso lo use.

Todos los objetos de Java tienen un monitor. Esta característica se encuentra integrada en el propio lenguaje de Java. La sincronización es soportada por la palabra clave **synchronized** y por unos cuantos métodos bien definidos que todos los objetos tienen. Debido a que la sincronización estuvo diseñada en Java desde el principio, es mucho más fácil usarla de lo que se podría esperar al principio. En realidad, para muchos programas la sincronización de objetos es casi transparente.

Existen dos maneras en las que puede sincronizar su código. Ambas incluyen el uso de la palabra clave **synchronized** y se examinarán a continuación.

HABILIDAD FUNDAMENTAL 11.8

Uso de métodos sincronizados

Puede sincronizar el acceso a un método si lo modifica con la palabra clave **synchronized**. Cuando se llama a ese método, el subproceso que llama entra en el monitor del objeto, que entonces bloquea al objeto. Mientras esté bloqueado, ningún otro subproceso puede entrar en el método, o entrar en cualquier otro método sincronizado que haya sido definido por el objeto. Cuando el subproceso regresa del método, el monitor desbloquea el objeto, permitiendo que el siguiente subproceso lo use. La sincronización se alcanza casi sin que usted realice un mayor esfuerzo de programación.

El siguiente programa demuestra la sincronización al controlar el acceso a un método llamado **sumaMatriz()**, que suma los elementos de una matriz de enteros.

```
// Uso de synchronize para controlar el acceso.

class SumaMatriz {
  private int suma;

  synchronized int sumMatriz(int nums[]) {
```

← **SumMatriz()** está sincronizada.

```
    suma = 0; // restablece la suma

    for(int i=0; i<nums.length; i++) {
      suma += nums[i];
      System.out.println("La ejecución total del " +
             Thread.currentThread().getName() +
             " es " + suma);
      try {
        Thread.sleep(10); // permite interrupción de tarea
      }
      catch(InterruptedException exc) {
        System.out.println("Subproceso principal interrumpido.");
      }
    }
    return suma;
  }
}

class MiSubproceso implements Runnable {
  Thread subpr;
  static SumaMatriz sm = new SumaMatriz();
  int a[];
  int respuesta;

  // Construye un nuevo subproceso.
  MiSubproceso(String nombre, int nums[]) {
    subpr = new Thread(this, nombre);
    a = nums;
    subpr.start(); // inicia el subproceso
  }

  // Empieza la ejecución del nuevo subproceso.
  public void run() {
    int suma;

    System.out.println(subpr.getName() + " iniciando.");

    respuesta = sm.sumMatriz(a);
    System.out.println("La suma de " + subpr.getName() +
                       " es " + respuesta);

    System.out.println(subpr.getName() + " terminando.");
  }
}

class Sync {
```

```
  public static void main(String args[]) {
    int a[] = {1, 2, 3, 4, 5};

    MiSubproceso ms1 = new MiSubproceso("descendiente #1", a);
    MiSubproceso ms2 = new MiSubproceso("descendiente #2", a);
  }
}
```

La salida del programa se muestra a continuación. (Es posible que la salida precisa sea diferente en su computadora.)

```
descendiente #1 iniciando.
La ejecución total del descendiente #1 es 1
descendiente #2 iniciando.
La ejecución total del descendiente #1 es 3
La ejecución total del descendiente #1 es 6
La ejecución total del descendiente #1 es 10
La ejecución total del descendiente #1 es 15
La suma de descendiente #1 es 15
descendiente #1 terminando.
La ejecución total del descendiente #2 es 1
La ejecución total del descendiente #2 es 3
La ejecución total del descendiente #2 es 6
La ejecución total del descendiente #2 es 10
La ejecución total del descendiente #2 es 15
La suma de descendiente #2 es 15
descendiente #2 terminando.
```

Examinemos con detalle este programa. Éste crea tres clases: la primera es **SumaMatriz**, la cual contiene el método **sumaMatriz( )** que suma una matriz de enteros. La segunda clase es **MiSubproceso**, que usa un objeto de tipo **SumaMatriz** para obtener la suma de una matriz de enteros. Por último, la clase **Sync** crea dos subprocesos y hace que éstos calculen la suma de una matriz de enteros.

Dentro de **sumaMatriz( )**, se llama a **sleep( )** para permitir que un cambio de tarea ocurra, en caso de que pueda presentarse (aunque no es el caso). Debido a que **sumaMatriz( )** está sincronizado, sólo puede ser usado por un subproceso al mismo tiempo. Por lo tanto, cuando un segundo subproceso descendiente inicia la ejecución, no entra en **sumaMatriz( )** sino hasta después de que el primer subproceso descendiente haya terminado con él. Esto asegura que el resultado correcto se produzca.

Para comprender por completo los efectos de **synchronized**, trate de eliminarlo de la declaración de **sumaMatriz( )**. Después de hacerlo, **sumaMatriz( )** ya no estará sincronizado, por lo que cualquier número de subprocesos podrán utilizarlo de manera concurrente. El problema es que la ejecución total está almacenada en suma, la cual cambiará con cada subproceso que llama a **sumaMatriz( )**. Por lo tanto, cuando dos subprocesos llamen a **sumaMatriz( )** al mismo tiempo, se producirán resultados incorrectos, ya que suma refleja la sumatoria de ambos subprocesos mezclados. Por ejemplo, he aquí una salida de ejemplo del programa después de que se ha eliminado **synchronized** de la declaración de **sumaMatriz( )**. (La salida precisa puede diferir en su computadora.)

```
Descendiente #1 iniciando.
La ejecución total del descendiente #1 es 1
descendiente #2 iniciando.
La ejecución total del descendiente #2 es 1
La ejecución total del descendiente #1 es 3
La ejecución total del descendiente #2 es 5
La ejecución total del descendiente #2 es 8
La ejecución total del descendiente #1 es 11
La ejecución total del descendiente #2 es 15
La ejecución total del descendiente #1 es 19
La ejecución total del descendiente #2 es 24
La suma de descendiente #2 es 24
descendiente #2 terminando.
La ejecución total del descendiente #1 es 29
La suma de descendiente #1 es 29
descendiente #1 terminando.
```

Como la salida muestra, ambos subprocesos descendientes usan **sumaMatriz()** de manera concurrente y el valor de suma está corrompido.

Antes de seguir adelante, revisemos los puntos clave de un método sincronizado.

- Se crea un método sincronizado si se antecede su declaración con **synchronized**.
- Para cualquier objeto determinado, una vez que se ha llamado al método sincronizado, el objeto queda bloqueado y ningún otro subproceso de ejecución puede usar métodos sincronizados en el mismo objeto.
- Otros subprocesos que traten de llamar a un objeto sincronizado en uso entrarán en estado de espera hasta que el objeto quede desbloqueado.
- Cuando un subproceso deja al método sincronizado, el objeto queda desbloqueado.

HABILIDAD FUNDAMENTAL 11.9

La instrucción synchronized

Aunque la creación de métodos **synchronized** dentro de las clases creadas constituye una manera fácil y efectiva de alcanzar la sincronización, no funcionará en todos los casos. Por ejemplo, tal vez quiera sincronizar el acceso a algún método que no esté modificado por **synchronized**. Esto ocurre porque quizá quiere usar una clase que no fue creada por usted sino por un tercero y no tiene acceso al código fuente. En consecuencia, no puede agregar **synchronized** a los métodos apropiados dentro de la clase. ¿De qué manera el acceso a un objeto de está clase puede ser sincronizado? Por fortuna, la solución a este problema es muy fácil. Simplemente ponga llamadas a los métodos definidos por esta clase dentro del bloque de **synchronized**.

Ésta es la forma general del bloque **synchronized**:

```
synchronized(objeto) {
// instrucciones que se sincronizarán
}
```

Aquí, *objeto* es una referencia al objeto que se está sincronizando. Un bloque sincronizado asegura que una llamada a un método que es un miembro de *objeto* sólo tendrá lugar después de que se ha ingresado el monitor del objeto en el subproceso que llama.

Por ejemplo, otra manera de sincronizar llamadas a **sumaMatriz()** consiste en llamar a éste desde el interior de un bloque sincronizado, como se muestra en esta versión del programa.

```
// Uso de un bloque sincronizado para controlar el acceso a sumaMatriz.
class SumaMatriz {
  private int suma;

  int sumaMatriz(int nums[]) {
    sum = 0; // reset suma

    for(int i=0; i<nums.length; i++) {
      suma += nums[i];
      System.out.println("La ejecución total de " +
             Thread.currentThread().getName() +
             " es " + suma);
      try {
        Thread.sleep(10); // permite la interrupción de tarea
      }
      catch(InterruptedException exc) {
        System.out.println("Subproceso principal interrumpido.");
      }
    }
    return suma;
  }
}

class MiSubproceso implements Runnable {
  Thread subpr;
  static SumaMatriz sm = new SumaMatriz();
  int a[];
  int respuesta;
```

Aquí, **sumaMatriz()** no está sincronizada.

```
  // Construye un nuevo subproceso.
  MiSubproceso(String nombre, int nums[]) {
    subpr = new Thread(this, nombre);
    a = nums;
    subpr.start(); // inicia el subproceso
  }

  // Empieza la ejecución de un nuevo subproceso.
  public void run() {
    int suma;

    System.out.println(subpr.getName() + " iniciando.");

    // sincroniza las llamadas a sumaMatriz()
    synchronized(sm) {
      respuesta = sm.sumaMatriz(a);
    }
    System.out.println("La suma de " + subpr.getName() +
                       " es " + respuesta);

    System.out.println(subpr.getName() + " terminando.");
  }
}

class Sync {
  public static void main(String args[]) {
    int a[] = {1, 2, 3, 4, 5};

    MiSubproceso ms1 = new MiSubproceso("descendiente #1", a);
    MiSubproceso ms2 = new MiSubproceso("descendiente #2", a);

    try {
      ms1.subpr.join();
      ms2.subpr.join();
    } catch(InterruptedException exc) {
      System.out.println("Subproceso principal interrumpido.");
    }
  }
}
```

Aquí están sincronizadas las llamadas a **sumaMatriz()** en sm.

Esta versión produce la misma salida correcta que la que se mostró antes y que usaba un método sincronizado.

Comprobación de avance

1. ¿Cómo se establece la prioridad de un subproceso?
2. ¿Cómo se restringe el acceso a un objeto a un subproceso a la vez?
3. La palabra clave **synchronized** puede usarse para modificar un método o para crear un bloque _____.

HABILIDAD FUNDAMENTAL

11.10 Comunicación entre subprocesos empleando notify(), wait() y notifyAll()

Imagine la siguiente situación: un subproceso llamado S se está ejecutando dentro de un método sincronizado y necesita acceso a recursos llamados R que no están disponibles temporalmente. ¿Qué debe hacer S? Si entra en alguna especie de bucle que espera a R, S se une al objeto evitando que otros subprocesos tengan acceso a él. Esta solución está lejos de ser óptima porque desactiva parcialmente las ventajas de programar para un entorno de subprocesos múltiples. Una mejor solución consiste en hacer que S abandone temporalmente el control del objeto, permitiendo que otro subproceso se ejecute. Cuando R queda disponible, es posible notificar a S, el cual reanuda su ejecución. Este método depende de cierta forma de comunicación entre subprocesos en la que un subproceso puede notificar a otro que está bloqueado y recibir una notificación de que puede reanudar la ejecución. Java soporta la comunicación entre subprocesos mediante los métodos **wait()**, **notify()** y **notifyAll()**.

Los métodos **wait()**, **notify()** y **notifyAll()** forman parte de todos los objetos ya que están implementados por la clase **Object**. Estos métodos sólo pueden llamarse dentro de un método **synchronized**. He aquí cómo se usan: cuando un subproceso tiene temporalmente bloqueada su ejecución, llama a **wait()**, lo que origina que el subproceso se detenga y que el monitor de ese objeto se libere, permitiendo que otro subproceso use el objeto. En este último punto, el subproceso que se encuentra en descanso se despierta cuando algún otro subproceso entra en el mismo monitor y llama a **notify()** o **notifyAll()**. Una llamada a **notify()** reanuda un subproceso, en tanto que una llamada a **notifyAll()** reanuda todos los subprocesos. Así, el subproceso con la mayor prioridad obtiene el acceso a ese objeto.

A continuación se presentan las diversas formas de **wait()** definidas por **Object**.

1. Para establecer la prioridad de un subproceso, debe llamar a **setPriority()**.
2. Para restringir el acceso a un objeto a un subproceso a la vez, debe usar la palabra **synchronized**.
3. **synchronized**.

final void wait() throws InterruptedException

final void wait(long *milisegundos*) throws InterruptedException

final void wait(long *milisegundos*, int *nanos*) throws InterruptedException

La primera forma espera hasta que recibe una notificación. La segunda forma espera hasta que se le notifica o hasta que el periodo especificado de milisegundos expira. La tercera forma le permite especificar el periodo de espera en nanosegundos.

He aquí las formas generales de **notify()** y **notifyAll()**.

final void notify()

final void notifyAll()

Un ejemplo que utiliza wait() y notify()

Para comprender la necesidad de **wait()** y **notify()**, así como su aplicación, crearemos un programa que simule el tic-tac de un reloj al desplegar las palabras "**tic**" y "**tac**" en la pantalla. Para realizarlo, crearemos una clase llamada Tictac que contenga dos métodos: **tic()** y **tac()**. El método **tic()** despliega la palabra "Tic" y **tac()** despliega "Tac". Para ejecutar el reloj, se han creado dos subprocesos, uno que llame a **tic()** y otro que llame a **tac()**. El objetivo consiste en hacer que los dos subprocesos se ejecuten de manera que la salida del programa despliegue un "Tic Tac" consistente, es decir, un patrón repetido de un tic seguido de un tac.

```
// Uso de wait() y notify() para crear un reloj que haga tic-tac.

class TicTac {

  synchronized void tic(boolean ejecutando) {
    if(!ejecutando) { // detiene el reloj
      notify(); // subprocesos de notificación y espera
      return;
    }

    System.out.print("Tic ");
    notify(); // que se ejecuta tac()
    try {
      wait(); // espera a que se complete tac()
    }
    catch(InterruptedException exc) {
      System.out.println("Thread interrumpido.");
    }
  }

  synchronized void tac(boolean ejecutando) {
```

tic() espera a **tac()**.

```
    if(!ejecutando) { // detiene el reloj
      notify(); // subprocesos de notificación y espera
      return;
    }

    System.out.println("Tac");
    notify(); // que se ejecute tic()
    try {
      wait(); // espera a que se complete tic
    }
    catch(InterruptedException exc) {
      System.out.println("Subproceso interrumpido.");
    }
  }
}

class MiSubproceso implements Runnable {
  Thread subpr;
  TicTac ttOb;

  // Construye un nuevo subproceso.
  MiSubproceso(String nombre, TicTac tt) {
    subpr = new Thread(this, nombre);
    ttOb = tt;
    subpr.start(); // inicia el subproceso
  }

  // Empieza la ejecución del nuevo subproceso.
  public void run() {

    if(subpr.getName().compareTo("Tic") == 0) {
      for(int i=0; i<5; i++) ttOb.tic(true);
      ttOb.tic(false);
    }
    else {
      for(int i=0; i<5; i++) ttOb.tac(true);
      ttOb.tac(false);
    }
  }
}

class SubprocesoCom {
  public static void main(String args[]) {
    TicTac tt = new TicTac();
    MiSubproceso ms1 = new MiSubproceso("Tic", tt);
    MiSubproceso ms2 = new MiSubproceso("Tac", tt);
```

Y **tac()** espera a **tic()**.

Llama a **tic()** y **tac()** a través de dos subprocesos diferentes.

```
    try {
      ms1.subpr.join();
      ms2.subpr.join();
    } catch(InterruptedException exc) {
      System.out.println("Subproceso principal interrumpido.");
    }
  }
}
```

He aquí la salida producida por el programa:

```
Tic Tac
Tic Tac
Tic Tac
Tic Tac
Tic Tac
```

Echemos un vistazo de cerca a este programa. En **main()**, se crea un objeto de **Tictac** llamado **tt**, el cual se usa para iniciar dos subprocesos de ejecución. Dentro del método **run()** de **MiSubproceso**, si el nombre del subproceso es "Tic", entonces se hacen llamadas a **tic()**. Si el nombre del subproceso es "Tac", entonces se llama al método **tac()**. Se hacen, a cada método, cinco llamadas que pasan **true** como argumento. El reloj se ejecuta siempre y cuando se pase **true**. Una llamada final que pase **false** a cada método detiene el reloj.

La parte más importante del programa se encuentra en los métodos **tic()** y **tac()**. Empezaremos con el método **tic()**, que, por conveniencia, se muestra a continuación:

```
synchronized void tic(boolean ejecutando) {
  if(!ejecutando) { // detiene el reloj
    notify(); // subprocesos de notificación y espera
    return;
  }

  System.out.print("Tic ");
  notify(); // que se ejecuta tac()
  try {
    wait(); // espera a que se complete tac()
  }
  catch(InterruptedException exc) {
    System.out.println("Thread interrumpido.");
  }
}
```

En primer lugar, note que **tic()** es modificado por **synchronized**. Recuerde que **wait()** y **notify()** sólo se aplican a los métodos sincronizados. El método empieza por revisar el valor del parámetro

ejecutando. Este parámetro se usa para proporcionar un cierre limpio del reloj. Si es **false**, entonces el reloj se detiene. Si éste es el caso, se hace una llamada a **notify()** para permitir que cualquier otro subproceso se ejecute. Más adelante regresaremos a este punto. Suponiendo que el reloj se está ejecutando cuando **tic()** se ejecuta, la palabra "Tic" se despliega; y luego tiene lugar una llamada a **notify()**, seguida de una llamada a **wait()**. La llamada a **notify()** permite que un subproceso que espera al mismo objeto se ejecute. La llamada a **wait()** hace que **tic()** se suspenda hasta que otro subproceso llame a **notify()**. Por consiguiente, cuando **tic()** es llamado, despliega un "Tic", deja que otro subproceso se ejecute y luego queda en suspenso.

El método **tac()** es una copia exacta de **tic()**, con la excepción de que despliega "Tac". En consecuencia, cuando se entra en él, despliega "Tac", llama a **notify()** y luego espera. Cuando se ve como un par, una llamada a **tic()** sólo puede ser seguida de una llamada a **tac()**, la cual sólo puede continuar con una llamada a **tic()**, y así sucesivamente. De ahí que los dos métodos estén mutuamente sincronizados.

La llamada a **notify()** cuando el reloj se detiene permite que una llamada final a **wait()** tenga éxito. Recuerde que **tic()** y **tac()** ejecutan una llamada a **wait()** después de desplegar su mensaje. El problema es que cuando el reloj se detiene, uno de los métodos estará esperando todavía. Se requiere entonces una llamada final a **notify()** para que el método que está esperando se ejecute. Como experimento, trate de eliminar esta llamada a **notify()** y observe lo que sucede. Como verá, el programa se "colgará" y necesitará oprimir CONTROL-C para salir. La razón de ello es que cuando la llamada final a **tac()** llama a **wait()**, no hay una llamada correspondiente a **notify()** que permita que concluya **tac()**. Por lo tanto, **tac()** se quedará "sentado" allí, esperando para siempre.

Antes de seguir adelante, si tiene alguna duda acerca de si las llamadas a **wait()** y **notify()** son realmente necesarias o no para que el "reloj" se ejecute de manera correcta, sustituya el programa anterior con esta versión de **Tictac**. Este programa hace que se eliminen todas las llamadas a **wait()** y **notify()**.

```
// No hay llamadas a wait() o notify().
class TicTac {

  synchronized void tic(boolean ejecutando) {
    if(!ejecutando) { // detiene el reloj
      return;
    }

    System.out.print("Tic ");
  }

  synchronized void tac(boolean ejecutando) {
    if(!ejecutando) { // detiene el reloj
      return;
    }

    System.out.println("Tac");
  }
}
```

Después de la sustitución, la salida producida por el programa tendrá este aspecto:

```
Tic Tac Tic Tac Tic Tac
Tac
Tac
Tac
Tac
```

¡Claramente los métodos **tic()** y **tac()** ya no están sincronizados!

Comprobación de avance

1. ¿Cuáles métodos soportan la comunicación entre subprocesos?
2. ¿Todos los objetos soportan la comunicación entre subprocesos?
3. ¿Qué sucede cuando se llama a **wait()**?

Pregunte al experto

P: **He oído que el término** bloqueado **y** muerto **se aplica a programas de subprocesos múltiples que se comportan mal. ¿En qué consiste éste y cómo puedo evitarlo?**

R: Bloqueado y muerto, como su nombre implica, es la situación en la que un subproceso está esperando a que otro subproceso haga algo, mientras que este último está esperando al primero. En consecuencia, ambos subprocesos están suspendidos esperándose mutuamente, por lo que ninguno se ejecuta. ¡Esta situación es análoga a la insistencia de dos personas excesivamente corteses cuando ambos esperan a que uno de ellos entre primero por la puerta!

Aparentemente resulta fácil evitar esta situación, pero no es así. Por ejemplo, la situación de bloqueado y muerto puede presentarse de manera indirecta. Por lo general, no es posible comprender fácilmente la causa con sólo observar el código fuente del programa debido a que los subprocesos que se ejecutan de manera concurrente pueden interactuar de manera compleja en tiempo de ejecución. Para evitar la situación de bloqueado y muerto, se requiere una programación cuidadosa y una prueba completa. Recuerde que si los programas de subprocesos múltiples suelen "colgarse", la causa por lo general es el bloqueo y la muerte.

1. Los métodos para la comunicación entre subprocesos son **wait()**, **notify()** y **notifyAll()**.
2. Sí, todos los objetos soportan comunicación entre subprocesos porque este soporte es parte de **Object**.
3. Cuando **wait()** es llamado, el subproceso que llama renuncia al control del objeto y se suspende hasta que recibe una notificación.

HABILIDAD FUNDAMENTAL 11.11

Suspensión, reanudación y detención de subprocesos

En ocasiones resulta útil suspender la ejecución de un subproceso. Por ejemplo, puede emplearse un subproceso separado para desplegar la hora. Si el usuario no desea un reloj, entonces su subproceso puede suspenderse. Cualquiera que sea la razón, suspender un subproceso resulta sencillo. Una vez suspendido, reiniciar el subproceso también resulta sencillo.

El mecanismo para suspender, detener y reanudar los subprocesos difiere entre las versiones iniciales de Java y las versiones más modernas surgidas a partir de Java 2. Antes de Java 2, un programa usaba **suspend()**, **resume()** y **stop()**, que eran métodos definidos de **Thread**, para hacer una pausa, reiniciar y detener la ejecución de un subproceso. Estos métodos tienen las siguientes formas:

```
final void resume( )

final void suspend( )

final void stop( )
```

Aunque parecían una manera perfectamente razonable y conveniente de manejar la ejecución de subprocesos, estos métodos no deben usarse más. He aquí el por qué: el método **suspend()** de la clase **Thread** fue eliminado de Java 2 porque **suspend()** en ocasiones generaba serias fallas de sistema. Suponga que el subproceso ha obtenido bloqueos en estructuras de datos fundamentales. Si ese subproceso se suspende en este punto, no se deshace de esos bloqueos. Otros subprocesos que estuvieran esperando esos recursos podrían caer en una situación de bloqueo y muerte. El método **resume()** se desechó también: no ocasiona problemas pero no puede utilizarse sin el método **suspend()** como contraparte. El método **stop()** de la clase **Thread** también se dejó de lado en Java 2 pues este método podía también generar en ocasiones serias fallas del sistema.

Debido a que hoy día no puede usar los métodos **suspend()**, **resume()** o **stop()** para controlar un subproceso, tal vez al principio considere que no hay manera de detener, reanudar o terminar un subproceso. Por fortuna, esto no es cierto. En cambio, puede diseñarse un subproceso para que el método **run()** revise periódicamente y determine si ese subproceso debe suspenderse, reanudarse o detener su propia ejecución. Por lo general, esto se logra al establecer dos marcas variables: una para suspender y reanudar y una para detener. En el caso de suspender y reanudar, siempre y cuando la marca esté en "ejecutándose", el subproceso debe detenerse. En el caso de la marca de alto, si está en "stop", el subproceso debe terminar.

En el siguiente ejemplo se muestra la manera de implementar sus propias versiones de **suspend()**, **resume()** y **stop()**.

```
// Suspensión, reanudación y detención de un subproceso.

class MiSubproceso implements Runnable {
  Thread subpr;
  volatile boolean suspendido;
  volatile boolean detenido;

  MiSubproceso(String nombre) {
    subpr = new Thread(this, nombre);
    suspendido = false;
    detenido = false;
    subpr.start();
  }

  // Éste es el punto de entrada del subproceso.
  public void run() {
    System.out.println(subpr.getName() + " iniciando.");
    try {
      for(int i = 1; i < 1000; i++) {
        System.out.print(i + " ");
        if((i%10)==0) {
          System.out.println();
          Thread.sleep(250);
        }

        // Uso del bloque sincronizado para revisar suspendido y detenido.
        synchronized(this) {
          while(suspendido) {
            wait();
          }
          if(detenido) break;
        }
      }
    } catch (InterruptedException exc) {
      System.out.println(subpr.getName() + " interrumpido.");
    }
    System.out.println(subpr.getName() + " saliendo.");
  }

  // Detiene el subproceso.
  synchronized void mideten() {
    detenido = true;

   // Lo siguiente hace que un subproceso suspendido sea detenido.
    suspendido = false;
    notify();
```

Suspende el subproceso cuando es **true**.

Detiene el subproceso cuando es **true**.

Este bloque sincronizado revisa si se está **suspendido** o **detenido**.

```
  }

  // Suspende el subproceso.
  synchronized void misuspend() {
    suspendido = true;
  }

  // Reanuda el subproceso.
  synchronized void mireanud() {
    suspendido = false;
    notify();
  }
}

class Suspender {
  public static void main(String args[]) {
    MiSubproceso ob1 = new MiSubproceso("Mi Subproceso");

    try {
      Thread.sleep(1000); // que el  subproceso ob1 inicie la ejecución

      ob1.misuspend();
      System.out.println("Suspendiendo subproceso.");
      Thread.sleep(1000);

      ob1. mireanud();
      System.out.println("Reanudando subproceso.");
      Thread.sleep(1000);


      ob1.misuspend();
      System.out.println("Suspendiendo subproceso.");
      Thread.sleep(1000);

      ob1. mireanud();
      System.out.println("Reanudando subproceso.");
      Thread.sleep(1000);

      ob1.misuspend();
      System.out.println("Deteniendo subproceso.");
      ob1.mideten();
    } catch (InterruptedException e) {
      System.out.println("Subproceso principal interrumpido");
    }

    // Espera a que termine el subproceso
```

```
    try {
      ob1.subpr.join();
    } catch (InterruptedException e) {
      System.out.println("Subproceso principal interrumpido");
    }

    System.out.println("Subproceso principal saliendo.");
  }
}
```

A continuación se muestra la salida de ejemplo de este programa. (Su salida puede diferir ligeramente.)

```
Mi Subproceso iniciando.
1 2 3 4 5 6 7 8 9 10
11 12 13 14 15 16 17 18 19 20
21 22 23 24 25 26 27 28 29 30
31 32 33 34 35 36 37 38 39 40
Suspendiendo subproceso.
Reanudando subproceso.
41 42 43 44 45 46 47 48 49 50
51 52 53 54 55 56 57 58 59 60
61 62 63 64 65 66 67 68 69 70
71 72 73 74 75 76 77 78 79 80
Suspendiendo subproceso.
Reanudando subproceso.
81 82 83 84 85 86 87 88 89 90
91 92 93 94 95 96 97 98 99 100
101 102 103 104 105 106 107 108 109 110
111 112 113 114 115 116 117 118 119 120
Deteniendo subproceso.
Mi Subproceso saliendo.
Subproceso principal saliendo.
```

He aquí la manera en que funciona el programa: la clase de subproceso **miSubproceso** define dos variables booleanas: **supendido** y **detenido**, que rigen la suspensión y la terminación de un subproceso. El constructor inicializa ambos en **false**. El método **run()** contiene un bloque de instrucciones **synchronized** que revisa **suspendido**. Si esa variable es **true**, se invoca al método **wait()** para suspender la ejecución del subproceso. Para ello, llama a **misuspend()** que establece **suspendido** como **true**. Para reanudar la ejecución, llama a **mireanud()**, que asigna **false** a **suspendido** e invoca a **notify()** para reiniciar el subproceso.

Para detener el subproceso, se llama a **mideten()**, que establece **detenido** como **true**. Además, **mideten()** establece suspend en **false** y luego llama a **notify()**. Estos pasos son necesarios para detener un subproceso suspendido.

Pregunte al experto

P: **Los subprocesos múltiples parecen mejorar en gran medida la eficiencia de mis programas. ¿Podría aconsejarme sobre cómo usarlos de manera efectiva?**

R: La clave para utilizar de manera efectiva los subprocesos múltiples consiste en pensar de manera concurrente en lugar de serial. Por ejemplo, cuando dentro de un programa tiene dos subsistemas que son fuertemente independientes entre sí, tome en cuenta la opción de convertirlos en subprocesos individuales. Sin embargo, es necesario tomar algunas precauciones. Si crea demasiados subprocesos, en realidad puede degradar el desempeño de su programa en lugar de mejorarlo. Recuerde que la sobrecarga de trabajo está asociada con el cambio del contexto. Si crea demasiados subprocesos, ¡se gastará más tiempo de CPU en cambiar los contextos que en ejecutar su programa!

Una nota adicional acerca del programa anterior: observe que **suspendido** y **detenido** se encuentran antecedidos por la palabra clave **volatile**. El modificador **volatile** es otra de las palabras clave de Java, y se analizará en el módulo 14. En resumen, ésta le indica al compilador que otras partes del programa pueden cambiar una variable de manera inesperada (como otro subproceso).

Proyecto 11.2 Uso del subproceso principal

`UsoPrincipal.java`

Todos los programas de Java tienen al subproceso llamado *subproceso principal*, el cual es otorgado al programa automáticamente cuando comienza a ejecutarse. Hemos tomado el subproceso principal como garantía. En este proyecto, verá que el subproceso puede ser llevado como cualquier otro.

Paso a paso

1. Cree un archivo llamado **UsoPrincipal.java**.
2. Para acceder al subproceso principal, debe obtener un objeto de **Thread** que haga referencia a él. Para ello, debe llamar al método **currentThread( )**, que es un miembro static de **Thread**. Ésta es su forma general:

   static Thread currentThread( )

   Este método regresa una referencia al subproceso en el que se le llama. Por lo tanto, si llama a **currentThread( )** mientras la ejecución está dentro del subproceso principal, obtendrá una referencia a éste. Una vez que tenga esta referencia, puede controlar el subproceso principal de la misma manera que controla cualquier otro subproceso.

3. Ingrese el siguiente programa en el archivo. Este programa obtiene una referencia al subproceso principal, y luego obtiene y establece el nombre y la prioridad de éste.

```
/*
   Proyecto 11.2

  Control del subproceso principal.
*/

class UsoPrincipal {
  public static void main(String args[]) {
    Thread subpr;

    // Obtiene el subproceso principal.
    subpr = Thread.currentThread();

    // Despliega el nombre del subproceso principal.
    System.out.println("El subproceso principal se llama: " +
                       subpr.getName());

    // Despliega la prioridad del subproceso principal.
    System.out.println("Prioridad: " +
                       subpr.getPriority());

    System.out.println();

    // Establece el nombre y la prioridad.
    System.out.println("Estableciendo el nombre y la prioridad.\n");
    subpr.setName("Subproceso #1");
    subpr.setPriority(Thread.NORM_PRIORITY+3);

    System.out.println("El subproceso principal se llama ahora: " +
                       subpr.getName());

    System.out.println("La prioridad es ahora: " +
                       subpr.getPriority());
  }
}
```

4. Ésta es la salida del programa.

```
El subproceso principal se llama: main
Prioridad: 5
```

(continúa)

```
Estableciendo el nombre y la prioridad.

El subproceso principal se llama ahora: Subproceso #1
La prioridad es ahora: 8
```

5. Debe tener cuidado con las operaciones que realiza en el subproceso principal. Por ejemplo, si agrega el siguiente código al final de **main()**, el programa nunca terminará ¡porque seguirá esperando a que el subproceso principal termine!

```
try {
   subpr.join()
} catch(InterruptedException exc)
  System.out.println("Interrumpido")
}
```

Comprobación de dominio del módulo 11

1. ¿Por qué la capacidad de los subprocesos múltiples de Java le permite escribir programas más eficientes?
2. Los subprocesos múltiples están soportados por la clase _____ y la interfaz _____.
3. Cuando crea un objeto ejecutable, ¿por qué podría desear extender **Thread** en lugar de implementar **Runnable**?
4. Muestre cómo usar **join()** para esperar a que un objeto de subproceso llamado **MiSubpr** termine.
5. Muestre cómo establecer un subproceso llamado **MiSubpr** tres niveles por arriba de la prioridad normal.
6. ¿Qué efecto se obtiene al agregar la palabra clave **synchronized** a un método?
7. Los métodos **wait()** y **notify()** se usan para realizar _____.
8. Cambie la clase **Tictac** para que realmente lleve el tiempo. Es decir, haga que cada tic tome medio segundo y que cada tac tome otro medio segundo. De esta manera, cada tictac tomará un segundo. (No se preocupe por el tiempo que las tareas de cambio tomen.)
9. ¿Por qué no puede usar **suspend()**, **resume()** y **stop()** en nuevos programas?
10. ¿Qué método definido por **Thread** obtiene el nombre de un subproceso?
11. ¿Qué regresa **isAlive()**?
12. Por su cuenta, trate de agregar sincronización a la clase **Cola** desarrollada en módulos anteriores con el fin de que esté segura para su uso en subprocesos múltiples.

Módulo 12

Enumeraciones, autoencuadre e importación de miembros estáticos

HABILIDADES FUNDAMENTALES

Con el lanzamiento de J2SE 5 a finales de 2004, Java se expandió sustancialmente a partir de la adición de varias funciones nuevas del lenguaje. Estas adiciones modificaron de manera sustancial el carácter y el alcance del lenguaje. Las siguientes son las funciones agregadas por J2SE 5:

- Genéricos
- Enumeraciones
- Autoencuadre/desencuadre
- El bucle **for** mejorado
- Argumentos de longitud variable (varargs)
- Importación de miembros estáticos
- Metadatos (anotaciones)

Funciones como las enumeraciones y el autoencuadre/desencuadre responden a necesidades que habían surgido mucho tiempo atrás. Otros, como los genéricos y los metadatos, abrieron nuevos caminos. En ambos casos, estas nuevas funciones han cambiado profundamente a Java.

Dos de estas funciones, el bucle **for** mejorado y varargs, ya han sido analizadas. En este módulo se examinarán de manera detallada las enumeraciones, el autoencuadre y la importación de miembros estáticos. El módulo terminará con una revisión general de los metadatos y en el módulo 13 se analizarán los genéricos.

Si está usando una versión antigua de Java, anterior a la versión J2SE 5, no podrá usar las características descritas en éste y el siguiente módulo.

HABILIDAD FUNDAMENTAL 12.1

Enumeraciones

La enumeración es una función común de programación que se encuentra en muchos otros lenguajes de computación. Sin embargo, no formaba parte de la especificación original de Java debido en parte a que la enumeración resulta técnicamente conveniente, pero no constituye una necesidad. Sin embargo, con los años, muchos programadores han deseado que Java soporte enumeraciones porque éstas ofrecen una solución ingeniosa y estructurada a una variedad de tareas de programación. Esta solicitud fue concedida en la versión J2SE 5, en la cual se agregaron las enumeraciones a Java.

En su forma más simple, una *enumeración* es una lista de constantes con nombre que definen un nuevo tipo de datos. Un objeto de un tipo de enumeración puede contener sólo los valores que están definidos por la lista. Por lo tanto, una enumeración le proporciona una manera de definir con precisión un nuevo tipo de datos con un número fijo de valores válidos.

Las enumeraciones son comunes en la vida diaria. Por ejemplo, una enumeración de las monedas usadas en México son 10, 20 y 50 centavos, 1, 5, 10 y 20 pesos. Una enumeración de los meses del año incluye los nombres de enero a diciembre. Una enumeración de los días de la semana incluye domingo, lunes, martes, miércoles, jueves, viernes y sábado.

Desde la perspectiva de la programación, las enumeraciones son útiles cada vez que se requiere definir un conjunto de valores que representen una colección de elementos. Por ejemplo, podría usar una enumeración para representar un conjunto de códigos de estado, como éxito, espera, falla y reintento, que indiquen el avance de alguna acción. En el pasado, estos valores se definían como variables **final**, pero las enumeraciones ofrecen un método mucho más estructurado.

Fundamentos de las enumeraciones

Una enumeración se crea usando la nueva palabra clave **enum**. Por ejemplo, he aquí una enumeración simple que despliega varias formas de transporte:

```
// Una enumeración de transportes
enum Transporte {
  AUTO, CAMIONETA, AEROPLANO, TREN, BOTE
}
```

Los identificadores **AUTO**, **CAMIONETA**, etc, son denominados *constantes de enumeración*. Cada uno se declara implícitamente como un miembro público y estático de **Transporte**. Más aún, el tipo de las constantes de enumeración es el tipo de enumeración en el que están declaradas las constantes, que en este caso la constante es **Transporte**. En el lenguaje de Java, a estas constantes se les denomina de *tipo propio*. "Propio" alude a la enumeración incluida.

Una vez que ha definido una enumeración, puede crear una variable de ese tipo. Sin embargo, aunque las enumeraciones definen un tipo de clase, usted no crea una instancia de **enum** empleando **new**. En cambio, declara y usa la variable de enumeración de manera muy parecida a como lo hace con uno de los tipos primitivos. Por ejemplo, lo siguiente declara **tp** como una variable del tipo de enumeración **Transporte**.

```
Transporte tp;
```

Debido a que **tp** es del tipo **Transporte**, los únicos valores que pueden asignarse son los definidos por la enumeración. Por ejemplo, lo siguiente asigna a **tp** el valor **AEROPLANO**:

```
tp = Transporte.AEROPLANO;
```

Observe que el símbolo **AEROPLANO** está calificado por **Transporte**.

Es posible comparar dos constantes de enumeración empleando el operador relacional ==. Por ejemplo, esta instrucción compara el valor de **tp** con la constante **TREN**.

```
if(tp == Transport.train) // ...
```

También puede usarse un valor de enumeración para controlar una instrucción **switch**. Por supuesto, todas las instrucciones **case** deben usar constantes del mismo **enum** que las usadas por la expresión **switch**. Por ejemplo, este **switch** es perfectamente válido.

```
// Uso de enum para controlar una instrucción switch.
  switch(tp) {
    case AUTO:
      // ...
    case CAMIONETA:
      // ...
```

Observe que en las instrucciones **case**, los nombres de las constantes de enumeración se utilizan sin estar calificadas por el nombre de su tipo de enumeración. Es decir, se usa **CAMIONETA**, no **Transporte.CAMIONETA**. Esto se debe a que el tipo de enumeración en la expresión **switch** ya ha especificado implícitamente el tipo de **enum** de las constantes de **case**. No es necesario calificar las constantes en las instrucciones **case** con su nombre de tipo de **enum**. De hecho, si trata de hacerlo, un error de compilación se generará.

Cuando la constante de enumeración se despliega, como en la instrucción **println( )**, se da salida a su nombre. Por ejemplo, dada esta instrucción:

```
System.out.println(Transporte.BOTE):
```

se despliega el nombre **BOTE**.

El siguiente programa reúne todos los elementos y demuestra la enumeración **Transporte**.

```
// Una enumeración de variedades de transporte.
enum Transporte {
  AUTO, CAMIONETA, AEROPLANO, TREN, BOTE
}

class EnumDemo {
  public static void main(String args[])
  {
    Transporte tp;

    tp = Transporte.AEROPLANO;

    // Da salida a un valor de enum.
```

Declara una enumeración.

Declara una referencia a **Transporte**.

Asigna tp a la constante **AEROPLANO**.

```
    System.out.println("Valor de tp: " + tp);
    System.out.println();

    tp = Transporte.TREN;

    // Compara dos valores de enum.
    if(tp == Transporte.TREN)
      System.out.println("tp contiene TREN.\n");

    // Uso de enum para controlar una instrucción switch.
    switch(tp) {
      case AUTO:
        System.out.println("Un auto transporta gente.");
        break;
      case CAMIONETA:
        System.out.println("Una camioneta transporta carga.");
        break;
      case AEROPLANO:
        System.out.println("Un aeroplano vuela.");
        break;
      case TREN:
        System.out.println("Un tren corre sobre rieles.");
        break;
      case BOTE:
        System.out.println("Un bote navega en el agua.");
        break;
    }
  }
}
```

Compara la igualdad de dos objetos de **Transporte**.

Usa una enumeración para controlar una instrucción **switch**.

Ésta es la salida del programa:

```
Valor de tp: AEROPLANO

tp contiene TREN.

Un tren corre sobre rieles.
```

Antes de seguir adelante, es necesario destacar un tema relacionado con el estilo. Las constantes de **Trasporte** emplean mayúsculas (se usa **AUTO**, no **auto**.) Sin embargo, no es necesario el uso de mayúsculas. En otras palabras, no hay una regla que imponga que las constantes de enumeración estén en mayúsculas. Debido a que con frecuencia las enumeraciones reemplazan a las variables de **final**, que generalmente se han usado en mayúsculas, algunos programadores creen que también la inclusión en mayúsculas de las constantes de enumeración resulta apropiada. Por supuesto, hay otros puntos de vista y estilos. Los ejemplos de este libro usarán mayúsculas para las constantes de enumeración con el fin de mantener la consistencia.

Comprobación de avance

1. Una enumeración define una lista de constantes _____.
2. ¿Cuál palabra clave declara una enumeración?
3. Dado

```
enum Direcciones {
  IZQUIERDA, DERECHA, ARRIBA, ABAJO
}
```

¿Cuál es el tipo de datos de **ARRIBA**?

HABILIDAD FUNDAMENTAL 12.2

Las enumeraciones de Java son tipos de clases

Aunque en los ejemplos anteriores muestran la mecánica de creación y uso de una enumeración, no muestran todas sus capacidades. A diferencia de la manera en que las enumeraciones se implementan en muchos otros lenguajes, *Java implementa las enumeraciones como tipos de clase*. Aunque usted no crea una instancia de **enum** usando **new**, actúa en otros sentidos de manera similar a otras clases. El hecho de que **enum** defina una clase permite que las enumeraciones de Java tengan opciones que las enumeraciones en otros lenguajes carecen. Por ejemplo, puede proporcionarle constructores, agregar variables de instancia y métodos, e incluso implementar interfaces.

HABILIDAD FUNDAMENTAL 12.3

Los métodos values() y valueOf()

Todas las enumeraciones cuentan automáticamente con dos métodos predefinidos: **values()** y **valueOf()**. Éstas son sus formas generales:

public static *tipo-enum*[ ] values( )

public static *tipo-enum*[ ] valueOf(String *cad*)

El método **values()** regresa una matriz que contiene una lista de constantes de enumeración. El método **valueOf()** regresa la constante de enumeración cuyo valor corresponde a la cadena pasada en *cad*. En ambos casos, *tipo-enum* es el tipo de la enumeración. Por ejemplo, en el caso de la

1. con nombre.
2. **enum**
3. El tipo de datos de **ARRIBA** es **Direcciones** porque las constantes enumeradas son de tipo propio.

enumeración **Transporte** que se mostró antes, el tipo de regreso de **Transporte.valueOf("TREN")** es **Transporte**. El valor regresado es **TREN**.

El siguiente programa demuestra los métodos **values()** y **ValueOf()**.

```
// Uso de los métodos de enumeración integrados.

// Una enumeración de variedades de transporte.
enum Transporte {
  AUTO, CAMIONETA, AEROPLANO, TREN, BOTE
}

class EnumDemo2 {
  public static void main(String args[])
  {
    Transporte tp;

    System.out.println("He aquí todas las constantes de transporte");

    // uso de values()
    Transporte todosTransportes[] = Transporte.values();
    for(Transporte t : todosTransportes)
      System.out.println(t);

    System.out.println();

    // uso de valueOf()
    tp = Transporte.valueOf("AEROPLANO");
    System.out.println("tp contiene " + tp);

  }
}
```

Obtiene una matriz de constantes de **Transporte**.

Obtiene la constante con el nombre **AEROPLANO**.

Ésta es la salida del programa:

```
He aquí todas las constantes de transporte
AUTO
CAMIONETA
AEROPLANO
TREN
BOTE

tp contiene AEROPLANO
```

Observe que este programa usa un bucle **for** de estilo for-each para recorrer en ciclo la matriz de constantes obtenida al llamar a **values()**. A manera de ejemplo, se creó la variable **todosTransportes** y

se le asignó una referencia a la matriz de enumeración. Sin embargo, este paso no es necesario porque es posible escribir el **for** como se muestra aquí, con lo que se elimina la necesidad de la variable **todosTransportes**:

```
for(Transporte t : Trasnporte.values())
  System.out.println(t);
```

Ahora, observe cómo el valor correspondiente al nombre **AEROPLANO** se obtuvo al llamar **valueOf()**:

```
tp = Transporte.valueOf("AEROPLANO");
```

Como se explicó, **valueOf()** regresa el valor de enumeración asociado con el nombre de la constante que está representada como una cadena.

HABILIDAD FUNDAMENTAL 12.4

Constructores, métodos, variables de instancia y enumeraciones

Es importante comprender que cada constante de enumeración es un objeto de su tipo de enumeración. Por lo tanto, una enumeración puede definir constructores, agregar métodos y tener variables de instancia. Cuando usted define un constructor para un **enum**, se llama al constructor cuando se crea cada constante de enumeración. Cada una de estas constantes puede llamar a cualquier método definido por la enumeración. Cada constante de enumeración tiene su propia copia de cualquier variable de instancia definida por la enumeración. La siguiente versión de **Transporte** ilustra el uso de un constructor, una variable de instancia y un método. Esta versión le asigna a cada tipo de transporte una velocidad típica.

```
// Uso de un constructor, una variable de instancia y un método de enum.
enum Transporte {
  AUTO(90), CAMIONETA(75), AEROPLANO(960), TREN(110), BOTE(30);

  private int velocidad; // velocidad típica de cada transporte

  // Constructor
  Transporte(int v) { velocidad = v; }

  int obtenerVelocidad() { return velocidad; }
}

class EnumDemo3 {
  public static void main(String args[])
  {
    Transporte tp;
```

Observe los valores de inicialización.

Agrega una variable de instancia.

Agrega un constructor.

Agrega un método.

```
    // Despliega velocidad de un aeroplano.
    System.out.println("La velocidad típica de un aeroplano es " +
                       Transporte.AEROPLANO.obtenerVelocidad() +
                       " km por hora.\n");

    // Despliega todos los transportes y velocidades.
    System.out.println("Velocidades de todos los transportes: ");
    for(Transporte t : Transporte.values())
      System.out.println(t + " la velocidad típica es " +
                         t.obtenerVelocidad() +
                         " km por hora.");
  }
}
```

Obtiene la velocidad al llamar a **obtenerVelocidad()**.

Ésta es la salida:

```
La velocidad típica de un aeroplano es 960 km por hora.

Velocidades de todos los transportes:
AUTO la velocidad típica es 90 km por hora.
CAMIONETA la velocidad típica es 75 km por hora.
AEROPLANO la velocidad típica es 960 km por hora.
TREN la velocidad típica es 110 km por hora.
BOTE la velocidad típica es 30 km por hora.
```

Esta versión de **Transporte** agrega tres aspectos: el primero es la variable de instancia **velocidad**, la cual se usa para contener la velocidad de cada tipo de transporte. La segunda es el constructor de **Transporte**, al cual se le pasa la velocidad de un transporte. La tercera es el método **obtenerVelocidad()**, que regresa el valor de **velocidad**.

Cuando la variable **tp** se declara en **main()**, se llama al constructor de **Transporte** una vez para cada constante especificada. Observe cómo, al colocarlos entre paréntesis, se especifican los argumentos del constructor después de cada constante, como se muestra aquí:

```
AUTO(90), CAMIONETA(75), AEROPLANO(960), TREN(110), BOTE(30);
```

Estos valores se pasan al parámetro **v** de **Transporte()**, que luego asigna este valor a **velocidad**. Se llama una vez al constructor para cada constante. Note además que la lista de constantes de enumeración termina con un punto y coma. Es decir, la última constante, **BOTE**, es seguida por un punto y coma. Cuando una enumeración contiene otros miembros, la lista de números debe terminar con un punto y coma.

Debido a que cada constante de enumeración contiene su propia copia de **velocidad**, puede obtener la velocidad de un tipo específico de transporte al llamar a **obtenerVelocidad()**. Por ejemplo, en **main()** la velocidad de un aeroplano se obtiene con la siguiente llamada:

```
Transporte.AEROPLANO.obtenerVelocidad()
```

Pregunte al experto

P: **Ahora que las enumeraciones son parte de Java, ¿debo evitar el uso de las variables** final**?**

R: No, las enumeraciones resultan apropiadas cuando trabaja con listas de elementos que deben ser representadas por identificadores. Una variable **final** es apropiada cuando tiene un valor constante, como un tamaño de matriz, que se utilizará en muchos lugares. Por lo tanto, cada una tiene su propio uso. La ventaja de las enumeraciones es que ahora no es necesario forzar a las variables final para que funcionen en un trabajo para el cual no son del todo adecuadas.

La velocidad de cada transporte se obtiene al recorrer en ciclo la enumeración empleando un bucle **for**. Debido a que hay una copia de **velocidad** para cada constante de enumeración, el valor asociado con una constante se encuentra separado y es distinto del valor asociado con otra. Se trata de un concepto importante, que sólo está disponible cuando las enumeraciones se implementan como clases, como lo hace Java.

Aunque el ejemplo anterior contiene sólo un constructor, un **enum** puede ofrecer dos o más formas sobrecargadas al igual que cualquier otra clase.

Dos restricciones importantes

Se aplica dos restricciones a las enumeraciones: en primer lugar, una enumeración no puede heredar otra clase; en segundo lugar, una **enum** no puede ser una superclase, lo que significa que una **enum** no puede extenderse. De otra manera, **enum** actúa de forma muy parecida a cualquier otro tipo de clase. La clave está en recordar que cada una de las constantes de enumeración es un objeto de la clase en la que está definida.

HABILIDAD FUNDAMENTAL 12.5

Las enumeraciones heredan Enum

Aunque no puede heredar una superclase cuando declara una **enum**, todas las enumeraciones heredan automáticamente una: **java.lang.Enum**. Esta clase define varios métodos que están disponibles para que todas las enumeraciones los utilicen. Lo más frecuente es que no utilice estos métodos; sin embargo, hay dos que ocasionalmente empleará: **ordinal( )** y **compareTo( )**.

El método **ordinal( )** obtiene un valor que indica la posición de una constante de enumeración en la lista de constantes. A éste se le llama *valor ordinal*. El método **ordinal( )** se muestra aquí:

```
final int ordinal( )
```

Regresa el valor ordinal de la constante que invoca. Los valores ordinales empiezan en cero; así que, en la enumeración **Transporte**, **AUTO** tiene un valor ordinal de cero; **CAMIONETA** tiene uno de 1; y **AEROPLANO** uno de 2, etcétera.

Tiene la opción de comparar el valor ordinal de dos constantes de la misma enumeración empleando el método **compareTo( )**, el cual tiene esta forma general:

final int compareTo(*tipo-enum e*)

Aquí, *tipo-enum* es el tipo de enum y *e* es la constante que se está comparando con la constante que invoca. Recuerde que la constante que invoca y *e* deben provenir de la misma enumeración. Si la constante que invoca tiene un valor ordinal menor que el de *e*, entonces **compareTo( )** regresa un valor negativo. Si los valores ordinales son iguales, entonces se regresa cero. Si la constante que invoca tiene un valor ordinal mayor que *e*, entonces se regresa un valor positivo.

El siguiente programa demuestra **ordinal( )** y **compareTo( )**.

```
// Demuestra ordinal() y compareTo().

// Una enumeración de variedades de transporte.
enum Transporte {
  AUTO, CAMIONETA, AEROPLANO, TREN, BOTE
}

class EnumDemo4 {
  public static void main(String args[])
  {
    Transporte tp, tp2, tp3;

    // Obtiene todos los valores ordinales usando ordinal().
    System.out.println("He aquí todas las constantes de Transporte" +
                       " y sus valores ordinales: ");
    for(Transporte t : Transporte.values())
      System.out.println(t + " " + t.ordinal()); <-- Obtiene valores ordinales.

    tp =  Transporte.AEROPLANO;
    tp2 = Transporte.TREN;
    tp3 = Transporte.AEROPLANO;

    System.out.println();

    // Demuestra compareTo()
    if(tp.compareTo(tp2) < 0) <-- Compara valores ordinales.
      System.out.println(tp + " llega antes que " + tp2);

    if(tp.compareTo(tp2) > 0)
```

```
      System.out.println(tp2 + " llega antes que " + tp);

    if(tp.compareTo(tp3) == 0)
      System.out.println(tp + " es igual a " + tp3);
  }
}
```

Ésta es la salida de este programa:

```
He aquí todas las constantes de Transporte y sus valores ordinales:
AUTO 0
CAMIONETA 1
AEROPLANO 2
TREN 3
BOTE 4

AEROPLANO llega antes que TREN
AEROPLANO es igual a AEROPLANO
```

Comprobación de avance

1. ¿Qué regresa **values()**?
2. ¿Una enumeración puede tener un constructor?
3. ¿Cuál es el valor ordinal de una constante de enumeración?

Proyecto 12.1 Un semáforo controlado por computadora

`SemáforoDemo.java`

Las enumeraciones son particularmente útiles cuando su programa necesita un conjunto de constantes. No obstante, los valores reales de las constantes son arbitrarios, a menos que todos sean diferentes. Este tipo de situación surge con mucha frecuencia cuando se crea un programa. Un ejemplo común incluye el manejo de los estados que un dispositivo puede tomar. Por ejemplo, imagine que está escribiendo un programa que controla un semáforo. Su código de semáforo debe recorrer en ciclo los tres estados de la luz: verde, amarillo y rojo. También debe permitir que otro código conozca el color actual de la luz y dejar que el color se establezca en un

1. El método **values()** regresa una matriz que contiene una lista de todas las constantes definidas por la enumeración que invoca.
2. Sí.
3. El valor ordinal de una constante de enumeración describe su posición en la lista de constantes; la primera constante tiene el valor ordinal de cero.

valor inicial conocido. Esto significa que los tres estados deben representarse de alguna manera. Aunque sería posible representar estos tres estados con valores enteros (por ejemplo, 1, 2 y 3), o con cadenas (como "rojo", "verde" y "amarillo"), una enumeración ofrece un método mucho mejor. El empleo de una enumeración arroja como resultado un código que, por un lado, es más eficiente que si los estados se representaran con cadenas y que, por el otro lado, es más estructurado que si se representaran con enteros.

En este proyecto, creará la simulación de un semáforo automatizado con base en la descripción anterior. Con este proyecto no sólo se muestra una enumeración en acción, también se muestra otro ejemplo de subprocesos múltiples y sincronización.

Paso a paso

1. Cree un archivo llamado **SemáforoDemo.java**

2. Empiece por definir una enumeración llamada **ColorSemáforo** que represente los tres estados de la luz, como se muestra aquí:

```
// Una enumeración de los colores de un semáforo.
enum ColorSemáforo {
  ROJO, VERDE, AMARILLO
}
```

   Cada vez que se necesite el color de la luz, se utilizará su valor de número.

3. A continuación, empiece a definir **SimuladorSemáforo** como se muestra a continuación. **SimuladorSemáforo** es la clase que encapsula la simulación del semáforo.

```
// Un semáforo computarizado.
class SimuladorSemáforo implements Runnable {
  private Thread subpr; // contiene el subproceso que ejecuta la simulación
  private ColorSemáforo cs; // contiene el color real del semáforo
  boolean alto = false; // se pone en true para detener la simulación

  SimuladorSemáforo(ColorSemáforo init) {
    cs = init;

    subpr = new Thread(this);
    subpr.start();
  }

  SimuladorSemáforo() {
    cs = ColorSemáforo.ROJO;

    subpr = new Thread(this);
    subpr.start();
  }
```

(continúa)

Observe que **SimuladorSemáforo** implementa **Runnable**, lo cual es necesario porque se emplea un subproceso separado para ejecutar cada luz. Este subproceso recorrerá en ciclo los colores. Se crean dos constructores: el primero le permite especificar el color inicial de la luz y el segundo se pone en rojo como opción predeterminada. Ambos empiezan un nuevo subproceso para ejecutar la luz.

Ahora observe las variables de instancia. Una referencia al subproceso de semáforo se almacena en **subpr**; el color real del semáforo se almacena en **cs**; y la variable **alto** se utiliza para detener la simulación (inicialmente se encuentra en false). La luz se ejecutará hasta que esta variable tome el valor true.

4. A continuación, agregue el método **run()**, el cual se muestra enseguida. Éste empieza a ejecutar el semáforo.

```
// Inicia la luz.
public void run() {
  while(!alto) {

    try {
      switch(cs) {
        case VERDE:
          Thread.sleep(10000); // verde por 10 segundos
          break;
        case AMARILLO:
          Thread.sleep(2000);  // amarillo por 2 segundos
          break;
        case ROJO:
          Thread.sleep(12000); // rojo por 12 segundos
          break;
      }
    } catch(InterruptedException exc) {
      System.out.println(exc);
    }
    cambiarColor();
  }
}
```

Este método recorre los colores del semáforo. En primer lugar, se mantiene suspendido, con base en el color actual, durante un tiempo apropiado. Luego, llama a **cambiarColor()** para cambiar al siguiente color de la secuencia.

5. Ahora, agregue el método **cambiarColor()** como se muestra aquí:

```
// Cambia el color.
synchronized void cambiarColor() {
  switch(cs) {
    case ROJO:
      cs = ColorSemáforo.VERDE;
      break;
    case AMARILLO:
```

```
        cs = ColorSemáforo.ROJO;
        break;
      case VERDE:
       cs = ColorSemáforo.AMARILLO;
    }

    notify(); // indica que la luz ha cambiado
  }
```

La instrucción **switch** examina el color almacenado actualmente en **cs** y luego asigna el siguiente color de la secuencia. Observe que este método está sincronizado, lo cual es necesario porque se llama a **notify()** para señalar que se ha producido un cambio de color. (Recuerde que sólo puede llamarse a **notify()** desde un método sincronizado.)

6. El siguiente método es **esperaCambio()**, el cual espera hasta que el color de la luz cambia.

```
// Espera hasta que un cambio de luz ocurre.
synchronized void esperaCambio() {
  try {
    wait(); // espera el cambio de luz
  } catch(InterruptedException exc) {
    System.out.println(exc);
  }
}
```

Este método simplemente llama a **wait()**. Esta llamada no regresará hasta que **cambiarColor()** ejecute una llamada a **notify()**. Por consiguiente, **esperaCambio()** no regresará hasta que el color haya cambiado.

7. Por último, agregue el método **obtenerColor()**, que devuelve el color actual de la luz, y **cancel()**, que detiene el subproceso del semáforo al establecer **alto** en true. A continuación se muestran estos métodos:

```
// Regresa el color real.
ColorSemáforo obtenerColor() {
  return cs;
}

// Detiene el semáforo.
void cancel() {
  alto = true;
}
```

8. He aquí todo el código ensamblado en un programa completo que demuestra el semáforo:

```
// Una simulación de un semáforo que utiliza
// una enumeración para describir el color de la luz.

// Una enumeración de los colores de un semáforo.
```

(continúa)

```
enum ColorSemáforo {
  ROJO, VERDE, AMARILLO
}

// Un semáforo computarizado.
class SimuladorSemáforo implements Runnable {
  private Thread subpr; // contiene el subproceso que ejecuta la simulación
  private ColorSemáforo cs; // contiene el color real del semáforo
  boolean alto = false; // se pone en true para detener la simulación

   SimuladorSemáforo(ColorSemáforo init) {
    cs = init;

    subpr = new Thread(this);
    subpr.start();
  }

  SimuladorSemáforo() {
    cs = ColorSemáforo.ROJO;

    subpr = new Thread(this);
    subpr.start();
  }

  // Inicia la luz.
  public void run() {
    while(!alto) {

      try {
        switch(cs) {
          case VERDE:
            Thread.sleep(10000); // verde por 10 segundos
            break;
          case AMARILLO:
            Thread.sleep(2000);  // amarillo por 2 segundos
            break;
          case ROJO:
            Thread.sleep(12000); // rojo por 12 segundos
            break;
        }
      } catch(InterruptedException exc) {
        System.out.println(exc);
      }
      cambiarColor();
    }
  }
```

```
  // Cambia el color.
  synchronized void cambiarColor() {
    switch(cs) {
      case ROJO:
        cs = ColorSemáforo.VERDE;
        break;
      case AMARILLO:
        cs = ColorSemáforo.ROJO;
        break;
      case VERDE:
       cs = ColorSemáforo.AMARILLO;
    }

    notify(); // indica que la luz ha cambiado
  }

  // Espera hasta que un cambio de luz ocurre.
  synchronized void esperaCambio() {
    try {
      wait(); // espera el cambio de luz
    } catch(InterruptedException exc) {
      System.out.println(exc);
    }
  }

  // Regresa el color real.
  ColorSemáforo obtenerColor() {
    return cs;
  }

  // Detiene el semáforo.
  void cancel() {
    alto = true;
  }
}

class SemáforotDemo {
  public static void main(String args[]) {
    SimuladorSemáforo tl = new SimuladorSemáforo(ColorSemáforo.VERDE);

    for(int i=0; i < 9; i++) {
      System.out.println(tl.obtenerColor());
      tl.esperaCambio();
    }
```

(continúa)

```
      tl.cancel();
    }
  }
```

Se produce la siguiente salida. Como puede ver, la luz del semáforo recorre en ciclo los colores en el orden de verde, amarillo y rojo.

```
VERDE
AMARILLO
ROJO
VERDE
AMARILLO
ROJO
VERDE
AMARILLO
ROJO
```

En el programa, observe cómo el uso de la enumeración simplifica y agrega estructura al código que necesita conocer el estado del semáforo. Debido a que la luz sólo puede tener tres estados (rojo, verde o amarillo), usar una enumeración asegura que sólo esos valores sean válidos, lo que evita un mal uso.

9. Es posible mejorar el programa anterior si se aprovechan las opciones de clase de una enumeración. Por ejemplo, si agrega un constructor, una variable de instancia y un método a **ColorSemáforo**, puede mejorar sustancialmente la programación anterior. Esta mejora se dejará como ejercicio. Vea la pregunta 4 de la Comprobación de dominio.

Autoencuadre

Con el lanzamiento de J2SE 5, Java agregó dos funciones muy deseadas por los programadores de Java: *autoencuadre* y *auto-desencuadre*. Esta función simplifica y delinea el código que debe convertir tipos primitivos en objetos y viceversa. Debido a que estas situaciones se encuentran con frecuencia en el código de Java, los beneficios del autoencuadre/desencuadre afectan a casi todos los programadores de Java. Como verá en el módulo 13, el autoencuadre/desencuadre contribuye en gran medida a la utilidad de otra función: los genéricos. La adición del autoencuadre/desencuadre cambia sutilmente la relación entre objetos y los tipos primitivos. Estos cambios son más profundos de los que la simplicidad conceptual del autoencuadre/desencuadre podría sugerir al principio. Sus efectos se sienten ampliamente en todo el lenguaje de Java.

El autoencuadre/desencuadre está directamente relacionado con los envolventes de tipo de Java, y con la manera en que los valores entran y salen de la instancia de un envolvente. Por ello, empezaremos con una revisión general de los envolventes de tipo y el proceso de encuadrar y desencuadrar valores manualmente.

HABILIDAD FUNDAMENTAL 12.6

Envolventes de tipo

Como ya lo sabe, Java emplea tipos primitivos, como **int** o **double**, para contener los tipos de datos básicos que el lenguaje soporta. Los tipos primitivos, en lugar de los objetos, se usan para estas cantidades por razones de desempeño. El uso de objetos para estos tipos básicos agregaría una sobrecarga inaceptable incluso en los cálculos más simples. Por lo tanto, los tipos primitivos no forman parte de la jerarquía de objetos y no heredan **Object**.

A pesar de los beneficios en desempeño que los tipos primitivos ofrecen, en ocasiones necesitará la representación de un objeto. Por ejemplo, no puede pasar un tipo primitivo por referencia a un método. Además, muchas de las estructuras de datos estándar implementadas por Java operan sobre objetos, lo que significa que no puede usar esas estructuras de datos para almacenar tipos primitivos. Para manejar estas (y otras) situaciones, Java proporciona *envolventes de tipo*, que son clases que encapsulan un tipo primitivo dentro de un objeto. Las clases de envolventes de tipos se introdujeron brevemente en el módulo 10. A continuación las estudiaremos más de cerca.

Los envolventes de tipo son **Double**, **Float**, **Long**, **Integer**, **Short**, **Byte**, **Character** y **Boolean**, que están empaquetados en **java.lang**. Estas clases ofrecen una amplia serie de métodos que le permiten integrar completamente los tipos primitivos en la jerarquía de objetos de Java.

Los envolventes de tipo de uso más común son en gran medida los que representan valores numéricos. Éstos son **Byte**, **Short**, **Integer**, **Long**, **Float** y **Double**. Todos los envolventes de tipo numérico heredan la clase abstracta **Number**. **Number** declara métodos que regresan el valor de un objeto en cada uno de los diferentes tipos numéricos. A continuación se muestran estos tipos:

```
byte byteValue( )

double doubleValue( )

float floatValue( )

int intValue( )

long longValue( )

short shortValue( )
```

Por ejemplo, **doubleValue( )** regresa el valor de un objeto como **double**, **floatValue( )** regresa el valor como **float**, etc. Cada uno de los envolventes de tipo numérico implementa estos métodos.

Todos los envolventes de tipo numérico definen constructores que permiten que un objeto sea construido a partir de un valor determinado o una representación de cadena de ese valor. Por ejemplo, he aquí los constructores definidos para **Integer** y **Double**:

Integer(int *num*)

Integer(String *cad*)

Double(double *num*)

Double(String *cad*)

Si *cad* no contiene un valor numérico válido, entonces se lanza una **NumberFormatException**.

Todos los envolventes de tipo sobrescriben **toString()**. Éste regresa la forma legible del valor contenido dentro del envolvente con el fin de que los seres humanos puedan leerlo. Esto le permite, por ejemplo, dar salida al valor al momento de pasar un objeto de envolvente de tipo a **println()**sin tener que convertirlo a su tipo primitivo.

Al proceso de encapsulamiento de un valor dentro de un objeto se le denomina *encuadre*. Antes de J2SE 5, todo el encuadre se llevaba a cabo manualmente, y los programadores llaman explícitamente a un método en un envolvente a fin de obtener su valor. Por ejemplo, lo siguiente desencuadra manualmente el valor de **iOb** en un **int**.

En este ejemplo, un nuevo objeto **int** con valor 100 es explícitamente creado y una referencia a este objeto es asignada a **iOb**.

El proceso de extraer un valor de un tipo de envolvente, se llama desencuadre. Otra vez, antes de J2SE 5, todo el desencuadre se lleva a cabo manualmente, y los programadores llaman explícitamente a un método en un envolvente para obtener este valor. Por ejemplo, éste desencuadra manualmente el valor en **iOb** en un **int**.

```
int i = iOb.initValue();
```

Aquí, **initValue()** devuelve el valor encapsulado dentro de **iOb** como **int**.

El siguiente programa demuestra los conceptos anteriores.

```
// Demuestra el encuadre y desencuadre manual con un envolvente de tipo.
class Envolvente {
  public static void main(String args[]) {

    Integer iOb = new Integer(100); ← Encuadra manualmente el valor 100.

    int i = iOb.intValue(); ← Desencuadra manualmente el valor en iOb.
```

```
    System.out.println(i + " " + iOb); // despliega 100 100
  }
}
```

Este programa envuelve el valor entero 100 dentro de un objeto de **Integer** llamado **iOb**. El programa obtiene entonces este valor al llamar a **intValue( )** y almacena el resultado en **i**. Por último, despliega los valores de **i** e **iOb**, que son 100.

Desde la versión original de Java se ha usado el mismo procedimiento general que se empleó en el ejemplo anterior para encuadrar y desencuadrar manualmente valores. Aunque este método para encuadrar y desencuadrar funciona, resulta tedioso y propenso a errores ya que requiere que el programador cree manualmente el objeto apropiado para envolver un valor y para obtener explícitamente el tipo primitivo apropiado cuando se requiere su valor. Por fortuna, en J2SE 5 se mejoraron de manera fundamental estos procedimientos esenciales mediante la adición del encuadre y el desencuadre.

HABILIDAD FUNDAMENTAL

12.7 Fundamentos del autoencuadre

El autoencuadre es el proceso mediante el cual un tipo primitivo es encapsulado automáticamente (encuadrado) en su envolvente de tipo equivalente, cada vez que se necesita un objeto de ese tipo. No hay necesidad de construir explícitamente un objeto. El autodesencuadre es el proceso mediante el cual el valor de un objeto encuadrado se extrae automáticamente (desencuadre) de un envolvente de tipo cuando se requiere su valor. No es necesario llamar a un método como **intValue( )** o **doubleValue( )**.

La adición del autoencuadre y autodesencuadre delinea de manera importante la codificación de varios algoritmos, eliminando el tedio de encuadrar y desencuadrar valores manualmente. Asimismo, contribuye a evitar errores. Con el autoencuadre ya no es necesario construir manualmente un objeto para envolver un tipo primitivo. Sólo necesita asignar ese valor a una referencia de envolvente de tipo. Java construye automáticamente el objeto por usted. He aquí, por ejemplo, la manera moderna de construir un objeto de **Integer** que tiene el valor de 100:

```
Integer iOb = 100;  // autoencuadra un int
```

Observe que no se crea explícitamente ningún objeto mediante el uso de **new**. Java lo lleva a cabo por usted de manera automática.

Para desencuadrar un objeto, simplemente asigne esa referencia a objeto a una variable de tipo primitivo. Por ejemplo, para desencuadrar **iOb**, puede usar esta línea:

```
int i = iOb;  // autodesencuadre
```

Java maneja los detalles por usted.

El siguiente programa demuestra las instrucciones anteriores.

```
// Demuestra el autoencuadre/desencuadre.
class AutoEncuadre {
  public static void main(String args[]) {

    Integer iOb = 100; // autoencuadra un int

    int i = iOb; // auto-desencuadra

    System.out.println(i + " " + iOb);  // despliega 100 100
  }
}
```

Autoencuadra y luego autodesencuadra el valor 100.

Comprobación de avance

1. ¿Cuál es el envolvente de tipo para **double**?
2. ¿Qué ocurre cuando encuadra un valor primitivo?
3. Autoencuadre es la característica que encuadra automáticamente un valor primitivo en un objeto de su envolvente de tipo correspondiente. ¿Cierto o falso?

HABILIDAD FUNDAMENTAL 12.8

Autoencuadre y métodos

Además de presentarse en un caso simple de asignaciones, el autoencuadre ocurre automáticamente cada vez que un tipo primitivo debe convertirse en un objeto, mientras que el autodesencuadre ocurre cada vez que un objeto debe convertirse en un tipo primitivo. Por consiguiente, el autoencuadre/desencuadre podría ocurrir al momento de pasar un argumento a un método o cuando un método regresa un valor. Por ejemplo, considere el siguiente código:

```
// El autoencuadre/desencuadre tiene lugar con
// parámetros de método y regresa valores.

class AutoEncuadre2 {
  // Este método tiene un parámetro Integer.
  static void m(Integer v) {
    System.out.println("m() recibió " + v);
  }
```

Recibe un **Integer**.

1. **Double**
2. Cuando un valor primitivo se encuadra, su valor se coloca dentro de un objeto de su envolvente de tipo correspondiente.
3. Cierto.

```
  // Este método regresa un int.
  static int m2() {
    return 10;
  }

  // Este método regresa un Integer.
  static Integer m3() {
    return 99; // autoencuadra 99 en un Integer.
  }

  public static void main(String args[]) {

    // Pasa un int a m().  Como m() tiene un parámetro Integer
    // parameter, el valor  int pasado se encuadra automáticamente.
    m(199);

    // Aquí, iOb recibe el valor int regresado por m2().
    // Este valor se encuadra automáticamente para que pueda
    // asignarse a iOb.
    Integer iOb = m2();
    System.out.println("El valor de regreso de m2() es " + iOb);

    // Ahora, se llama a m3(). Regresa un valor Integer
    // que se autoencuadra en un int.
    int i = m3();
    System.out.println("El valor de regreso de m3() es " + i);

    // En seguida, se llama a Math.sqrt() con iOb como argumento.
    // En este caso, iOb se autoencuadra y su valor se promueve a
    // double, que es el tipo necesario para  sqrt().
    iOb = 100;
    System.out.println("La raíz cuadrada de iOb es " + Math.sqrt(iOb));
  }
}
```

Recibe un **int**.

Regresa un **Integer**.

Este programa despliega el siguiente resultado:

```
m() recibió 199
El valor de regreso de m2() es 10
El valor de regreso de m3() es 99
La raíz cuadrada de iOb es 10.0
```

En el programa, observe que **m()** especifica un parámetro **Integer**. Dentro de **main()**, **m()** pasa el valor **int** 199. Debido a que **m()** está esperando un **Integer**, este valor se encuadra automáticamente. A continuación, se llama a **m2()**, el cual regresa el valor **int** de 10. Este valor **int** es asignado a **iOb** en **main()**. Debido a que **iOb** es un **Integer**, el valor regresado por **m2()** se autoencuadra. Enseguida

se llama a **m3()**, el cual regresa un **Integer** que se autodesencuadra dentro de un **int**. Por último, se llama a **Math.sqrt()** con **iOb** como argumento. En este caso, **iOb** se autodesencuadra y su valor se promueve a **double**, ya que es el tipo esperado por **Math.sqrt()**.

HABILIDAD FUNDAMENTAL 12.9

El autoencuadre/desencuadre ocurre en expresiones

En general, el autoencuadre y desencuadre ocurre cada vez que se requiere una conversión en un objeto o desde un objeto. Esto se aplica a expresiones. Dentro de una expresión, un objeto numérico se autoencuadra automáticamente. De ser necesario, la salida de la expresión se reencuadra. Por ejemplo, considere el siguiente programa:

```
// El autoencuadre/desencuadre ocurre dentro de expresiones.

class AutoEncuadre3 {
  public static void main(String args[]) {

    Integer iOb, iOb2;
    int i;

    iOb = 99;
    System.out.println("Valor original de iOb: " + iOb);

    // Lo siguiente desencuadra automáticamente iOb,
    // lleva a cabo un incremento y luego reencuadra
    // el resultado en iOb.
    ++iOb;
    System.out.println("Después de ++iOb: " + iOb);

    // Aquí, iOb es desencuadrado, su valor aumenta 10 y
    // el resultado se encuadra y se almacena otra vez en iOb.
    iOb += 10;
    System.out.println("Después de iOb += 10: " + iOb);

    // Aquí, se desencuadra iOb, se evalúa la expresión
    // y el resultado se reencuadra y se asigna
    // a iOb2.
    iOb2 = iOb + (iOb / 3);
    System.out.println("iOb2 después de la expresión: " + iOb2);

    // Se evalúa la misma expresión, pero el
    // resultado no se reencuadra.
    i = iOb + (iOb / 3);
```

El autoencuadre/desencuadre ocurre en expresiones.

```
    System.out.println("i después de la expresión: " + i);
  }
}
```

Aquí se muestra la salida:

```
Valor original de iOb: 99
Después de ++iOb: 100
Después de iOb += 10: 110
iOb2 después de la expresión: 146
i después de la expresión: 146
```

En el programa, preste atención especial a esta línea:

```
++iOb;
```

Esto hace que el valor de **iOb** se incremente. Funciona así: **iOb** se desencuadra, el valor se incrementa y el resultado se reencuadra.

Debido al autodesencuadre, puede usar objetos numéricos enteros, como **integer**, para controlar una instrucción **switch**. Por ejemplo, considere este fragmento:

```
Integer iOb = 2;

switch(iOb) {
  case 1: System.out.println("uno");
    break
  case 1: System.out.println("dos");
    break
  default: System.out.println("error");
}
```

Cuando se evalúa la expresión **switch**, **iOb** se desencuadra y se obtiene su valor **int**.

Como los ejemplos del programa lo muestran, debido al autoencuadre/desencuadre, el uso de objetos numéricos en una expresión resulta fácil e intuitivo. En el pasado, este código hubiera incluido moldeo y llamadas a métodos como **intValue()**.

Una palabra de advertencia

Ahora que Java incluye autoencuadre y autodesencuadre, tal vez se sienta tentado a usar exclusivamente objetos como **Integer** o **Double**, y con ello abandonar por completo los primitivos. Por ejemplo, con el autoencuadre/desencuadre es posible escribir un código como el siguiente:

```
// ¡Un mal uso de autoencuadre/desencuadre!
Double a, b, c;
```

```
a = 10.2;
b = 11.4;
c = 9.8;

Double prom = (a + b + c) / 3;
```

En este ejemplo, los objetos de tipo **Double** contienen valores que son promediados. El resultado se asigna a un objeto de **Double**. Aunque este código es técnicamente correcto y, de hecho, funciona de manera apropiada, no es recomendable usar autoencuadre/desencuadre pues es mucho menos eficiente que el código equivalente que se escribe empleando el tipo primitivo **double**. La razón es que cada autoencuadre y autodesencuadre agrega una sobrecarga que no estaría presente si se utilizara el tipo primitivo.

En general, debe restringir el uso de los envolventes de tipo y sólo emplearlos en los casos en que se necesita la representación de objeto de un tipo primitivo. El autoencuadre/desencuadre no se agregó a Java como una "puerta trasera" para eliminar los tipos primitivos.

Comprobación de avance

1. ¿Un valor primitivo se autoencuadra cuando se pasa como argumento a un método que está esperando un objeto de envolvente de tipo?
2. Debido a los límites impuestos por el sistema en tiempo de ejecución de Java, el autoencuadre/desencuadre no ocurrirá en los objetos usados en expresiones. ¿Cierto o falso?
3. Debido al autoencuadre/desencuadre, debe usar objetos en lugar de tipos primitivos para realizar la mayor parte de las operaciones aritméticas. ¿Cierto o falso?

HABILIDAD FUNDAMENTAL 12.10

Importación de miembros estáticos

J2SE 5 expandió el uso de la palabra clave **import** para que soporte una nueva función llamada *importación de miembros estáticos*. Al colocar después de **import** la palabra clave **static,** es posible utilizar una instrucción **import** para importar los miembros estáticos de una clase o interfaz. Cuando este tipo de importación se usa, es posible hacer referencia a miembros estáticos directamente por sus nombres, sin tener que calificarlos con el nombre de su clase. Esto simplifica y acorta la sintaxis que se requiere para usar un miembro estático.

1. Sí.
2. Falso.
3. Falso.

Para comprender la utilidad de la importación de miembros estáticos, empecemos con un ejemplo que *no la utiliza*. El siguiente programa calcula la solución de una ecuación cuadrática que tiene esta forma:

$ax^2 + bx + c = 0$

El programa usa dos métodos estáticos de la clase integrada de matemáticas de Java **Math**, la cual forma parte de **java.lang**. La primera es **Math.pow()**, que regresa un valor elevado a una potencia específica. El segundo es **Math.sqrt()**, que regresa la raíz cuadrada de su argumento.

```
// Encuentra la solución a una ecuación cuadrática.
class Cuadrática {
  public static void main(String args[]) {

    // a, b y c representan los coeficientes en la
    // ecuación cuadrática: ax² + bx + c = 0
    double a, b, c, x;

    // Resuelve 4x² + x - 3 = 0 para x.
    a = 4;
    b = 1;
    c = -3;

    // Encuentra la primera solución.
    x = (-b - Math.sqrt(Math.pow(b, 2) - 4 * a * c)) / (2 * a);
    System.out.println("Primera solución: " + x);

    // Encuentra la segunda solución.
    x = (-b - Math.sqrt(Math.pow(b, 2) - 4 * a * c)) / (2 * a);
    System.out.println("Segunda solución: " + x);
  }
}
```

Como **pow()** y **sqrt()** son métodos estáticos, deben llamarse mediante su nombre de clase, **Math**. Esto da como resultado una expresión que resulta un poco difícil de manejar:

```
x = (-b + Math.sqrt(Math.pow(b, 2) - 4 * a * c)) / (2 * a);
```

Más aún, al especificar el nombre de clase cada vez que se emplee **pow()** o **sqrt()** (o bien, cualquier otro método matemático de Java, como **sin()**, **cos()** y **tan()**), su uso puede volverse tedioso.

Puede eliminar el tedio de especificar el nombre de clase mediante el uso de la importación de miembros estáticos, como se muestra en la siguiente versión del programa anterior.

```
// Uso de la importación de miembros estáticos para traer a la vista sqrt()
y pow().

import static java.lang.Math.sqrt;
import static java.lang.Math.pow;

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

    // a, b y c representan los coeficientes en la
    // ecuación cuadrática: ax² + bx + c = 0
    double a, b, c, x;

    // Resuelve 4x² + x -3 = 0 para x.
    a = 4;
    b = 1;
    c = -3;

    // Encuentra la primera solución.
    x = (-b + sqrt(pow(b, 2) - 4 * a * c)) / (2 * a);
    System.out.println("Primera solución: " + x);

    // Encuentra la segunda solución.
    x = (-b - sqrt(pow(b, 2) - 4 * a * c)) / (2 * a);
    System.out.println("Segunda solución: " + x);
  }
}
```

Use la importación de miembros estáticos para traer **sqrt()** y **pow()** a la vista.

En esta versión, los nombres **sqrt** y **pow** se traen a la vista mediante las siguientes instrucciones de importación de miembros estáticos:

```
import static java.lang.Math.sqrt;
import static java.lang.Math.pow;
```

Después de estas instrucciones, ya no es necesario calificar a **sqrt()** o **pow()** con su nombre de clase. Por lo tanto, la expresión puede especificarse de manera más conveniente como se muestra aquí:

```
x = (-b + sqrt(pow(b, 2) - 4 * a * c)) / (2 * a);
```

Como verá, esta forma es considerablemente más corta y fácil de leer.

Hay dos formas generales de la instrucción **import static**. La primera, utilizada en el ejemplo anterior, trae a la vista un solo nombre. Esta es su forma general:

import static *paq.nombre-tipo.*; static-miembro-nombre;

Aquí, *nombre-tipo* es el nombre de una clase o interfaz que contiene el miembro estático deseado. Su nombre de paquete completo está especificado por *paq* y el nombre del miembro está especificado por *nombre-miembro-estático*.

La segunda forma importa todos los miembros estáticos. Aquí se muestra su forma general:

*import static paq.nombre-tipo.**;

Si requiere usar muchos métodos o campos estáticos definidos por una clase, entonces esta forma le permitirá traerlos a la vista sin tener que especificar cada uno de manera individual. Así, el programa anterior pudo haber utilizado únicamente la instrucción **import** para traer a **pow()** y **sqrt()** a la vista (y a *todos los demás* miembros estáticos de **Math**):

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

Por supuesto, la importación de miembros estáticos no está limitada solamente a la clase **Math** o a métodos. Por ejemplo, lo siguiente trae a la vista el campo estático **System.out**:

```
import static java.lang.System.out;
```

Después de esta instrucción, puede dar salida a la consola sin tener que calificar **out** con **System**, como se muestra aquí:

```
out.println("Después de importar System.out, puede usarlo directamente.");
```

Que la importación de **System.out,** como se acaba de mostrar, sea o no una buena idea es motivo de polémica. Si bien acorta la instrucción, no queda completamente claro, para cualquier persona que lea el programa, que **out** hace referencia a **System.out**.

Aunque la importación de miembros estáticos puede resultar muy conveniente, es importante no abusar de ella. Recuerde que la razón de que Java organice sus bibliotecas en paquetes es la de evitar las colisiones entre nombres de espacios. Cuando importa miembros estáticos, está trayendo esos miembros en el nombre de espacio global. Por lo tanto, aumenta el potencial de conflictos de nombre de espacio y el ocultamiento inadvertido de otros nombres. Si utiliza un miembro estático una o dos veces en el programa, es mejor no importarlo. Además, algunos nombres estáticos, como **System.out**, son tan reconocibles que tal vez no desee importarlos. La importación de miembros estáticos está diseñada para las situaciones en las que utiliza un miembro estático de manera repetida, como cuando realiza una serie de cálculos matemáticos. En esencia, debe usar esta función, pero no abusar de ella.

Pregunte al experto

P: **Mediante el uso de la importación de miembros estáticos, ¿puedo importarlos miembros estáticos de clases que cree?**

R: Sí, puede usar la importación de miembros estáticos para importar los miembros estáticos de clases e interfaces que usted cree. Llevar esto a cabo resulta especialmente conveniente cuando define varios miembros estáticos que se utilizan con frecuencia en todo un programa largo. Por ejemplo, si una clase define cierto número de constantes **static final** que definen varios límites, entonces el uso de la importación de miembros estáticos para traerlos a la vista le ahorrará en gran medida el tedio de escribirlos.

HABILIDAD FUNDAMENTAL 12.11

Metadatos

Entre las nuevas funciones agregadas a Java en J2SE 5, los metadatos son los más innovadores. Esta nueva y poderosa función le permite incrustar información complementaria en un archivo fuente. Esta información, llamada *anotación*, no cambia las acciones de un programa. Sin embargo, varias herramientas pueden usar esta información durante el desarrollo y el despliegue. Por ejemplo, una anotación podría ser procesada por un generador de código fuente, por el compilador, o por una herramienta de despliegue. Aunque Sun denomina a esta función metadatos, el término *función de anotación del programa* también se usa y probablemente sea más descriptivo.

Los metadatos constituyen un tema largo y complejo por lo que su estudio detallado se encuentra más allá del alcance de este libro. Sin embargo, se realizará una breve revisión general para que se familiarice con el concepto.

Podrá encontrar un análisis detallado de los metadatos y las anotaciones en el libro Java: The Complete Reference, J2SE 5 Edition (McGraw-Hill/Osborne, 2005).

Los metadatos se crean mediante un mecanismo basado en **interface**. He aquí un ejemplo simple:

```
// Un tipo de anotación simple
@interface MiAnot {
  String str();
  int val();
}
```

Esto declara una anotación llamada **MiAnot**. Observe la @ que antecede a la palabra clave **interface**. Esto le indica al compilador que un tipo de anotación se está declarando. A continuación, observe

los dos miembros **str()** y **val()**. Todas las anotaciones constan sólo de declaraciones de métodos. Sin embargo, usted no proporciona cuerpos para estos métodos, sino que Java los implementa. Más aún, los métodos actúan de manera parecida a los campos.

Todos los tipos de anotación extienden automáticamente la interfaz **Annotation**. Así, **Annotation** es una superinterfaz de todas las anotaciones y está declarada dentro del paquete **java.lang.annotation**.

Una vez que haya declarado una anotación, puede usarla para anotar una declaración. Cualquier tipo de declaración puede tener una anotación asociada. Por ejemplo, clases, métodos, campos, parámetros y constantes **enum** pueden estar anotados. Incluso una anotación puede anotarse. En todos los casos, la anotación antecede al resto de la declaración.

Al aplicar una anotación, usted proporciona valores a sus miembros. Por ejemplo, he aquí un ejemplo de **MiAnot** aplicado a un método:

```
// Anota un método.
@MiAnot(str = "Ejemplo de anotación", val = 100)
public static void MyMet() { // ...
```

Esta anotación está vinculada con el método **MiMet()**. Observe de cerca la sintaxis de la anotación. El nombre de la anotación, antecedido por una **@**, está seguido por una lista, entre paréntesis, de inicializaciones de miembros. Para proporcionar un valor a un miembro, a ese nombre se le asigna un valor. De ahí que, en el ejemplo, la cadena "Ejemplo de anotación" esté asignada al miembro **str** de **MiAnot**. Observe que en esta asignación no hay paréntesis después de **str**. Cuando un miembro de una anotación recibe un valor, sólo se usa su nombre. En consecuencia, en este contexto los miembros de la anotación parecen campos.

A las anotaciones que no tienen parámetros se les denomina *anotaciones de marcador*. Se les especifica sin pasar argumentos y sin usar paréntesis. Su único objetivo es marcar una declaración con algún atributo.

En este momento Java define siete anotaciones integradas: cuatro de ellas se importan de **java.lang.annotation**: **@Retention**, **@Documented**, **@Target** y **@Inherited**. Tres, **@Override**, **@Deprecated** y **@SuppressWarnings**, están incluidas en **java.lang**. Las anotaciones integradas se muestran en la tabla 12.1.

A continuación se muestran ejemplos que emplean **@Deprecated** para marcar la clase **MiClase** y el método **obtenerMsj()**. Cuando trate de compilar este programa, los avisos reportarán el uso de estos elementos desaprobados.

```
// Un ejemplo que usa @Deprecated.

// Desaprueba una clase.
@Deprecated ◄──────── Marca una clase como desaprobada.
class MiClase {
  private String msj;

  MiClase(String m) {
    msj = m;
```

```
  }

  // Desaprueba un método dentro de una clase.
  @Deprecated
  String obtenerMsj() {
    return msj;
  }

   // ...
}

class AnotDemo {
  public static void main(String args[]) {
    MiClase miObj = new MiClase("prueba");

    System.out.println(miObj.obtenerMsj());
  }
}
```

Marca un método como desaprobado.

Anotación	Descripción
@Retention	Especifica la política de retención que se asociará con la anotación. La política de retención determina el tiempo en el que la anotación estará presente durante la compilación y el proceso de despliegue.
@Documented	Una anotación de marcador que le indica a una herramienta que una anotación será documentada. Está diseñada para usarse sólo como una anotación de una declaración de anotación.
@Target	Especifica los tipos de declaraciones a los que puede aplicarse una declaración. Está diseñada para usarse sólo como una anotación de otra anotación. **@Target** toma un argumento, el cual debe ser una constante de la enumeración **ElementType**, que define varias constantes, como **CONSTRUCTOR**, **FIELD** y **METHOD**. El argumento determina los tipos de declaraciones a las que puede aplicarse la anotación.
@Inherited	Una anotación de marcador que hace que la anotación de una superclase sea heredada por una subclase.
@Override	Un método anotado con **@Override** debe sobrescribir un método de una superclase. Si no existe, se obtendrá un error en tiempo de compilación. Se utiliza para asegurar que un método de una superclase realmente se sobrescribe y no simplemente se sobrecarga. Es una anotación de marcador.
@Deprecated	Una anotación de marcador que indica que una declaración es obsoleta y ha sido reemplazada por una forma más reciente.
@SuppressWarnings	Especifica que es posible que uno o más avisos que el compilador podría emitir se suprimirán. Los avisos que se suprimirán se especifican por nombre, en forma de cadena.

Tabla 12.1 Las anotaciones integradas.

Comprobación de avance

1. Muestre las dos formas de importación de miembros estáticos.
2. Muestre cómo importar el método **sleep()** de **Thread** para que pueda utilizarse sin que **Thread** lo califique.
3. La importación de miembros estáticos funciona con métodos, pero no con variables. ¿Cierto o falso?
4. Una anotación empieza con una _____.

Comprobación de dominio del módulo 12

1. Se dice que las constantes de enumeración son de tipo propio. ¿Qué significa esto?
2. ¿Qué clases heredan automáticamente todas las enumeraciones?
3. Dada la siguiente enumeración, escriba un programa que use **values()** para mostrar una lista de constantes y sus valores ordinales.

```
enum Herramientas {
  DESARMADOR,  LLAVE, MARTILLO, PINZAS
}
```

4. La simulación del semáforo que se desarrolló en el proyecto 12.1 puede mejorarse con unos cuantos cambios que aprovechen las funciones de la clase de enumeración. En la versión mostrada, la duración de cada color estuvo controlada por la clase **SimuladorSemáforo** al codificar estos valores en el método **run()**. Cambie esto último para que la duración de cada color se almacene en las constantes de la enumeración **ColorSemáforo**. Para ello, necesitará agregar un constructor, una variable de instancia privada y un método llamado **obtenerDemora()**. Después de realizar estos cambios, ¿qué mejoras observa? Por su cuenta, ¿puede pensar en otras mejoras? (Sugerencia: trate de usar valores ordinales para cambiar los colores de las luces, en lugar de depender de la instrucción **switch**.)
5. Defina el encuadre y el desencuadre. ¿Cómo afecta el autoencuadre/desencuadre estas acciones?

1. import static *paq.nombre-tipo.nombre-miembro-estático*.
   import static *paq.nombre-tipo*.*;
2. import estatic java.lang.Thread.sleep;
3. Falso.
4. @

6. Cambie el siguiente fragmento con el fin de que éste emplee el autoencuadre.

```
Short val = new Short(123);
```

7. En sus propias palabras, ¿qué lleva a cabo la importación de miembros estáticos?

8. ¿Qué lleva a cabo esta instrucción?

```
import static java.lang.Integer.parseInt;
```

9. ¿La importación de miembros estáticos está diseñada para situaciones especiales, o es una buena práctica traer a la vista todos los miembros estáticos de todas las clases?

10. Una anotación está sintácticamente basada en _____.

11. ¿Qué es una anotación de marcador?

12. Una anotación puede aplicarse únicamente a métodos. ¿Cierto o falso?

Módulo 13

Elementos genéricos

HABILIDADES FUNDAMENTALES

Como se explicó en el módulo 12, recientemente se agregaron muchas funciones nuevas a Java y se incorporaron en la versión J2SE 5. Todas estas nuevas funciones mejoraron sustancialmente y expandieron el alcance del lenguaje, pero las que tienen un mayor impacto son los *elementos genéricos*, debido a que sus efectos se dejan sentir en todo el lenguaje Java. Por ejemplo, agregan un nuevo elemento de sintaxis y generan cambios a muchas de las clases y los métodos en el núcleo de la API. No resulta exagerado afirmar que la inclusión de los elementos genéricos ha modificado de manera fundamental la forma y el carácter de Java.

El tema de los elementos genéricos es muy extenso, así que en cierta forma resulta lo suficientemente avanzado como para en este libro no se abarque. Sin embargo, todos los programadores que empleen Java deben tener una comprensión básica de los elementos genéricos. A primera vista, la sintaxis de los elementos genéricos parece un poco intimidante, pero no se preocupe: su uso es sorprendentemente sencillo. Cuando termine este módulo, tendrá un dominio suficiente de los conceptos clave de los elementos genéricos, así como el conocimiento apropiado para usarlos de manera efectiva en sus propios programas.

Si está usando una versión antigua de Java que sea anterior a la versión J2SE 5, no podrá usar los elementos genéricos.

HABILIDAD FUNDAMENTAL 13.1

Fundamentos de los elementos genéricos

En su núcleo, el término *elementos genéricos* significa *tipos con parámetros*. Los tipos con parámetros son importantes porque le permiten crear clases, interfaces y métodos en los que el tipo de datos sobre los que operan está especificado como un parámetro. A la clase, interfaz o método que opera en un parámetro de tipo se le califica como *genérico* (por ejemplo, en los casos de *clase genérica* o *método genérico*).

Una ventaja importante del código genérico es que funcionará automáticamente con el tipo de datos que se pasan a su parámetro de tipo. Muchos algoritmos son lógicamente iguales, sin importar el tipo de datos a los que se están aplicando. Por ejemplo, un ordenamiento rápido es el mismo ya sea que los elementos de ordenamiento sean de tipo **Integer**, **String**, **Object** o **Thread**. Con los elementos genéricos, puede definir un algoritmo una vez, independientemente del tipo específico de datos, y luego aplicar este algoritmo a una amplia variedad de tipos de datos sin que se requiera un esfuerzo adicional.

Es importante comprender que Java siempre le ha brindado la capacidad de crear clases, interfaces y métodos generalizados al momento de operar mediante referencias de tipo a **Object**. Debido a que **Object** es una superclase de todas las demás clases, una referencia a **Object** puede hacer referencia a cualquier tipo de objeto. Por lo tanto, en el código anterior a los elementos genéricos, las clases, las interfaces y los métodos generalizados usaban referencias a **Object** para operar con varios tipos de datos. El problema era que no podían hacerlo con *seguridad de tipo* ya que se necesitaban moldeos para convertir explícitamente de **Object** al tipo real de datos sobre el que se estaba operando. En consecuencia, era posible crear, de manera accidental, tipos que no coincidían. Los elementos genéricos agregan la seguridad de tipo que hacía falta porque logran que estos moldeos sean automáticos e implícitos. En resumen, los elementos genéricos expanden su capacidad de reciclar código de manera segura y confiable.

Pregunte al experto

P: **He oído que los elementos genéricos de Java son similares a las plantillas de C++. ¿Es cierto?**

R: Los elementos genéricos de Java sí son similares a las plantillas de C++. Lo que en Java se denomina tipo con parámetros, en C++ se conoce como plantilla. Sin embargo, los genéricos de Java y las plantillas de C++ no son iguales, y existen diferencias fundamentales entre los dos acercamientos a los tipos genéricos. En su mayor parte, el uso del método de Java es más sencillo.

Una palabra de advertencia: si tiene conocimientos de C++, es importante que no saque conclusiones apresuradas acerca del funcionamiento de los elementos genéricos en Java. Los dos tratamientos del código genérico difieren en formas sutiles pero fundamentales.

HABILIDAD FUNDAMENTAL 13.2

Un ejemplo simple de elementos genéricos

Antes de analizar cualquier teoría adicional, es mejor buscar un ejemplo sencillo de elementos genéricos. El siguiente programa define dos clases: la primera es la clase genérica **Gen** y la segunda es **GenDemo**, la cual utiliza **Gen**.

```
// Una clase genérica simple.
// Aquí, T es un tipo con parámetros que
// será reemplazado por un tipo real
// cuando se cree un objeto de tipo Gen.
class Gen<T> {
  T ob; // declara un objeto de tipo T

  // Pasa el constructor una referencia a
  // un objeto de tipo T.
  Gen(T o) {
    ob = o;
 }

  // Regresa ob.
  T obtenerob() {
    return ob;
  }

  // Muestra el tipo de T.
  void mostrarTipo() {
    System.out.println("Type of T is " +
                       ob.getClass().getName());
  }
}
```

Declara una clase genérica. **T** es el parámetro de tipo genérico.

```
// Demuestra la clase genérica.
class GenDemo {
  public static void main(String args[]) {
    // Crea una referencia a Gen para Integer.
    Gen<Integer> iOb;

    // Crea un objeto Gen<Integer> y asigna su
    // referencia a iOb. Observe el uso de autoencuadre
    // para encapsular el valor 88 en un objeto Integer.
    iOb = new Gen<Integer>(88);

    // Muestra el tipo de datos usado por iOb.
    iOb.mostrarTipo();

    // Obtiene el valor de iOb. Observe que
    // no es necesario el moldeo.
    int v = iOb.obtenerob();
    System.out.println("Valor: " + v);

    System.out.println();

    // Crea un objeto Gen para Strings.
    Gen<String> cadOb = new Gen<String>("Prueba de genéricos");

    // Muestra el tipo de datos usado por cadOb.
    cadOb.mostrarTipo();

    // Obtiene el valor de cadOb. Una vez más, observe
    // que no es necesario el moldeo.
    String cad = cadOb.obtenerob();
    System.out.println("valor: " + cad);
  }
}
```

Crea una referencia a un objeto de tipo **Gen<Integer>**.

Crea una instancia de un objeto de tipo **Gen<Integer>**.

Crea una referencia y un objeto de tipo **Gen<String>**.

La salida producida por el programa se muestra aquí:

```
El tipo de T es java.lang.Integer
Valor: 88

El tipo de T es java.lang.String
valor: Prueba de genéricos
```

Examinemos con cuidado este programa. En primer lugar, observe cómo **Gen** es declarado en la siguiente línea:

```
class Gen<T> {
```

En este caso, **T** es el nombre de un *parámetro de tipo*. Este nombre se usa como marcador de posición para el tipo real que se pasará a **Gen** cuando se cree un objeto. Por consiguiente, **T** se usa dentro de **Gen** cada vez que se necesita el parámetro de tipo. Observe que **T** está contenido dentro de < >. Esta sintaxis puede generalizarse. Cada vez que declara un parámetro de tipo, se especifica dentro de paréntesis de pico. Debido a que **Gen** usa un parámetro de tipo, **Gen** es una clase genérica.

En la declaración de **Gen** no hay un significado especial para el nombre **T**, es decir, pudo emplearse cualquier identificador válido, pero **T** es tradicional. Más aún, se recomienda que los nombres de los parámetros de tipo sean letras mayúsculas de un solo carácter. Otros nombres de parámetros de tipo de uso común son **V** y **E**.

A continuación, **T** se usa para declarar un objeto llamado **ob**, como se muestra aquí:

```
T ob: // declara un objeto de tipo T
```

Como se explicó, T es una variable para el tipo actual que será especificado cuando un objeto **Gen** es creado. De este modo, **ob** será un objeto de tipo transformado a **T**. Por ejemplo, si el tipo **String** es transformado a **T**, después inmediatamente, **ob** será de tipo **String**. Ahora considere la construcción de **Gen**:

```
Gen(T o) {
  ob = o;
}
```

Observe que su parámetro, **o**, es de tipo **T**. Esto significa que el tipo real de **o** está determinado por el tipo pasado a **T** cuando el objeto de **Gen** se crea. Además, debido a que el parámetro **o** y la variable de miembro **ob** son de tipo **T**, serán del mismo tipo real cuando se cree el objeto de **Gen**.

El parámetro de tipo **T** también puede usarse para especificar el tipo de regreso del método, como en el caso del método **obtenerob( )** que se muestra a continuación:

```
T obtenerob() {
  return ob;
}
```

Debido a que **ob** también es de tipo **T**, su tipo es compatible con el tipo de regreso especificado por **obtenerob( )**.

El método **mostrarTipo( )** despliega el tipo de **T**, lo cual lleva a cabo al llamar a **getName( )** del objeto de **Class** regresado por la llamada a **getClass( )** en **ob**. No hemos utilizado antes esta función, de modo que debemos examinarla de cerca. Como seguramente recuerda del módulo 7, la clase **Object** define el método **getClass( )**, así que **getClass( )** es un miembro de todos los tipos de clase: regresa un objeto de **Class** que corresponde al tipo de clase del objeto en el que es llamado. **Class** es una clase definida dentro de **java.lang** que encapsula información acerca de una clase. **Class** define varios métodos

que pueden usarse para obtener información acerca de una clase en tiempo de ejecución. Entre éstos se encuentra el método **getName()**, que regresa una representación de cadena del nombre de la clase.

La clase **GenDemo** demuestra la clase genérica **Gen**. Primero crea una versión de **Gen** para enteros, como se muestra aquí:

```
Gen<Integer> iOb;
```

Observe cuidadosamente esta declaración. En primer lugar, observe que el tipo **Integer** está especificado dentro de picoparéntesis después de **Gen**. En este caso, **Integer** es un *argumento de tipo* que se pasa al parámetro de tipo **T** de **Gen**. Esto crea efectivamente una versión de **Gen** en la que todas las referencias a **T** se traducen en referencias a **Integer**. Por lo tanto, para esta declaración, **Object** es de tipo **Integer**, y el tipo de regreso de **obtenerob()** es de tipo **Integer**.

Antes de seguir adelante, es necesario establecer que en realidad el compilador de Java no crea versiones diferentes de **Gen**, o de cualquier otra clase genérica. En cambio, elimina toda la información de tipo genérico sustituyendo los moldeos necesarios para que su código se *comporte como si* se creara una versión específica de **Gen**. Así pues, en su programa sólo existe en realidad una versión de **Gen**. Al proceso de eliminar información de tipo genérico se le llama *borrado*, el cual se analizará más adelante en este módulo.

La siguiente línea asigna a **iOb** una referencia a una interfaz de una versión **Integer** de la clase **Gen**.

```
iOb = new Gen<Integer>(88);
```

Observe que cuando se llama al constructor de **Gen**, se especifica también el argumento de tipo **Integer**, lo cual es necesario porque el tipo del objeto (en este caso **iOb**) al que se está asignando la referencia es de tipo **Gen<Integer>**. Por tanto, la referencia regresada por **new** también debe ser de tipo **Gen<Integer>**. Si no lo es, arrojará como resultado un error en tiempo de compilación. Por ejemplo, la siguiente asignación causará un error en tiempo de compilación.

```
iOb = new Gen<Double>(88.0); // ¡Error!
```

Debido a que **iOb** es de tipo **Gen<Integer>**, no puede usarse para hacer referencia a un objeto de **Gen<Double>**. Esta comprobación de tipo constituye uno de los principales beneficios de los elementos genéricos puesto que confirma la seguridad de tipo.

Tal y como lo establece el comentario del programa, la asignación

```
iOb = new Gen<Integer>(88);
```

utiliza el autoencuadre para encapsular el valor 88, que es un **int**, en un **Integer**. Esto funciona porque **Gen<Integer>** crea un constructor que toma un argumento **Integer**. Debido a que se espera un

Integer, Java encuadra automáticamente 88 dentro de uno. Por supuesto, la asignación pudo haberse escrito explícitamente de esta manera:

```
iOb = new Gen<Integer>(new Integer(88));
```

Sin embargo, no se obtendría ningún beneficio al usar esta versión.

El programa despliega después el tipo de **ob** dentro de **iOb**, que es **Integer**. A continuación, el programa obtiene el valor de **Object** mediante el uso de la siguiente línea:

```
int v = iOb.obtenerob();
```

Debido a que el tipo de regreso de **obtenerob()** es **T**, que fue reemplazado con **Integer** cuando se declaró **iOb**, el tipo de regreso de **obtenerob()** también es **Integer**, el cual se autoencuadra en **int** cuando se asigna a **v** (que es un **int**). De modo que no es necesario moldear el tipo de regreso de **obtenerob()** a **Integer**.

A continuación, **GenDemo** declara un objeto de tipo **Gen<String>**:

```
Gen<String> cadOb = new Gen<String>("Prueba de genéricos");
```

Debido a que el argumento de tipo es **String**, se sustituye a **String** por **T** dentro de **Gen**. Como se demuestra en las líneas restantes del programa, esto crea (conceptualmente) una versión **String** de **Gen**.

Los elementos genéricos sólo funcionan con objetos

Cuando se declara una instancia de un tipo genérico, el argumento de tipo pasado al parámetro de tipo debe ser un tipo de clase. No puede usar un tipo primitivo, como **int** o **char**. Por ejemplo, con **Gen** es posible pasar cualquier tipo de clase a **T**, pero no es posible pasar un tipo primitivo a **T**. De manera que la siguiente declaración es ilegal:

```
Gen<int> cadOb = new Gen<int>(53); //  Error, no puede usar un tipo primitivo
```

Por supuesto, la incapacidad de especificar un tipo primitivo no constituye una restricción seria, ya que puede usar los envolventes de tipo (como se llevó a cabo en el ejemplo anterior) para encapsular un tipo primitivo. Más aún, el mecanismo de autoencuadre y autodesencuadre emplea el envolvente de tipo transparente.

Los tipos genéricos difieren con base en sus argumentos de tipo

Uno de los puntos clave para comprender los tipos genéricos es que la referencia de una versión específica de un tipo genérico no es compatible en tipo con otra versión del mismo tipo genérico. Por ejemplo, si toma en cuenta el programa que apenas se mostró, la siguiente línea de código es un error y no se compilará:

```
iOb = cadOb; // ¡Incorrecto!
```

Aunque **iOb** y **cadOb** son de tipo **Gen<T>**, representan referencias a diferentes tipos pues sus parámetros son diferentes. Esto es parte de la manera en que los elementos genéricos agregan seguridad de tipo y evitan errores.

Una clase genérica con dos parámetros de tipo

Puede declarar más de un parámetro de tipo en un tipo genérico. Para especificar dos o más parámetros de tipo, simplemente use una lista separada por comas. Por ejemplo, la siguiente clase **DosGen** es una variación de la clase **Gen** que cuenta con dos parámetros de tipo.

```
// Una clase genérica simple con dos tipos
// parámetros: T y V.
class DosGen<T, V> {  <-- Se usan dos parámetros de tipo.
  T ob1;
  V ob2;

  // Pasa el constructor una referencia a
  // un objeto de tipo T
  DosGen(T o1, V o2) {
    ob1 = o1;
    ob2 = o2;
  }

  // Muestra tipos de T y V.
  void mostrarTipos() {
    System.out.println("El tipo de T es " +
                       ob1.getClass().getName());

    System.out.println("El tipo de V es " +
                       ob2.getClass().getName());
  }

  T obtenerob1() {
    return ob1;
  }

  V obtenerob2() {
    return ob2;
  }
}

// Demuestra DosGen.
class SimpGen {
```

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

    DosGen<Integer, String> tgObj =
      new DosGen<Integer, String>(88, "Genéricos");

    // Muestra los tipos.
    tgObj.mostrarTipos();

    // Obtiene y muestra valores.
    int v = tgObj.obtenerob1();
    System.out.println("valor: " + v);

    String cad = tgObj.obtenerob2();
    System.out.println("valor: " + cad);
  }
}
```

Aquí, **Integer** se pasa a **T**, y **String** se pasa a **V**.

Ésta es la salida de este programa:

```
El tipo de T es java.lang.Integer
El tipo de V es java.lang.String
valor: 88
valor: Genéricos
```

Observe cómo se declara **DosGen**:

```
class DosGen<T, V> {
```

Especifica dos parámetros de tipo, **T** y **V**, separados por una coma. Debido a que cuenta con dos parámetros de tipo, deben pasarse dos argumentos de tipo a **DosGen** cuando se crea un objeto, como se muestra a continuación:

```
DosGen<Integer, String> tgObj =
  new DosGen<Integer, String>(88, "Genéricos");
```

En este caso, **Integer** es sustituido por **T** y **String** es sustituido por **V**.

Aunque los dos argumentos de tipo difieren en este ejemplo, es posible que ambos tipos sean iguales. Por ejemplo, la siguiente línea de código es válida:

```
DosGen<String, String> x = new DosGen<String, String>("A", "B");
```

En este caso, **T** y **V** serían de tipo **String**. Por supuesto, si el argumento de tipo fuera siempre el mismo, entonces dos parámetros de tipo resultarían innecesarios.

La forma general de una clase genérica

Es posible generalizar la sintaxis de los elementos genéricos que se muestra en los ejemplos anteriores. He aquí la sintaxis para declarar una clase genérica:

class *nombre-clase<lista-parámetro-tipo>* { // ...

He aquí la sintaxis para declarar una referencia a una clase de elementos genéricos:

nombre-clase<lista-argumentos-tipo> nombre-var =
new *nombre-clase<lista-argumentos-tipo>*(*lista-argumentos-cons*>;

Comprobación de avance

1. El tipo de datos operado bajo una clase genérica se pasa a ella a través de _____.
2. ¿Un parámetro de tipo puede pasar un tipo primitivo?
3. Considerando la clase **Gen** que se muestra en el ejemplo anterior, demuestre cómo declarar una referencia a **Gen** que opera en datos de tipo **Double**.

HABILIDAD FUNDAMENTAL 13.3

Tipos limitados

En los ejemplos anteriores, los parámetros de tipo podrían reemplazarse con cualquier tipo de clase, lo cual es correcto para varios fines. Sin embargo, en ocasiones resulta útil limitar los tipos que pueden pasarse a un parámetro de tipo. Por ejemplo, suponga que quiere crear una clase genérica que almacene un valor numérico y sea capaz de realizar varias funciones matemáticas, como calcular el recíproco u obtener el componente fraccionario. Más aún, quiere usar la clase para calcular estas cantidades para cualquier tipo de número, incluyendo enteros, flotantes y dobles. Por consiguiente, desea especificar el tipo de los números de manera genérica, usando un parámetro de tipo. Para crear este tipo de clase, podría probar algo como lo siguiente:

```
// Intento de FunsNumeric (sin éxito) de crear
// una clase genérica que calcule varias
// funciones numéricas, como el recíproco o el
```

1. Un parámetro de tipo.
2. No.
3. `Gen<Double> d_ob;`

```
// componente fraccionario, dado cualquier tipo de número.
class FunsNumeric<T> {
  T num;

  // Pasa el constructor una referencia a un
  // objeto numérico.
  FunsNumeric(T n) {
    num = n;
  }

  // Regresa el recíproco.
  double reciprocal() {
    return 1 / num.doubleValue(); // ¡Error!
  }

  // Regresa el componente fraccionario.
  double fraction() {
    return num.doubleValue() - num.intValue(); // ¡Error!
  }

  // ...
}
```

Por desgracia, **FunsNumeric** no se compilará como está escrito, ya que ambos métodos generarán errores en tiempo de compilación. Primero, examine el método **reciprocal( )** que trata de regresar el recíproco de **num**. Para ello, debe dividir 1 entre el valor de **num**. El valor de **num** se obtiene al llamar a **doubleValue( )**, el cual obtiene la versión **double** del objeto numérico almacenado en **num**. Debido a que todas las clases numéricas, como **Integer** y **Double**, son subclases de **Number**, y **Number** define el método **doubleValue( )**, este método está disponible para todas las clases de envolvente numérico. El problema es que para el compilador no es posible saber que usted está tratando de crear objetos de **FunsNumeric** usando sólo tipos numéricos. Por lo tanto, al intentar compilar **FunsNumeric**, se reporta un error que indica que el método **doubleValue( )** es desconocido. El mismo tipo de error ocurre dos veces en **fraction( )**, el cual necesita llamar a **doubleValue( )** y a **intValue( )**. Ambas llamadas dan como resultado mensajes de error que establecen que estos métodos son desconocidos. Para resolver este problema, necesita alguna manera de indicarle al compilador que usted intenta pasar sólo tipos numéricos a **T**. Más aún, necesita alguna manera de *asegurar* que *sólo* se pasan realmente tipos numéricos.

Para manejar estas situaciones, Java proporciona *tipos limitados*. Cuando especifique un parámetro de tipo, puede crear un límite superior que declare la superclase a partir de la cual deben derivarse los argumentos de tipo. Esto se logra mediante el uso de una cláusula **extends** al momento de especificar el parámetro de tipo, como se muestra aquí:

<*T* extends *superclase*>

Esto especifica que *T* sólo puede ser reemplazado por una *superclase*, o por subclases de la *superclase*. Por lo tanto, la *superclase* define un límite inclusivo y superior.

Puede usar un límite superior para corregir la clase **FunsNumeric** que se mostró antes si especifica **Number** como límite superior, como se muestra aquí:

```
// En esta versión de FunsNumeric, el argumento de tipo
// para T debe ser Number, o una clase derivada
// de Number.
class FunsNumeric<T extends Number> {
  T num;

  // Pasa al constructor una referencia a
  // un objeto numérico.
  FunsNumeric(T n) {
    num = n;
  }

  // Regresa el recíproco.
  double reciprocal() {
    return 1 / num.doubleValue();
  }

  // Regresa el componente fraccionario.
  double fraction() {
    return num.doubleValue() - num.intValue();
  }

  // ...
}

// Demuestra FunsNumeric.
class LímitesDemo {
  public static void main(String args[]) {

    FunsNumeric<Integer> iOb =
                      new FunsNumeric<Integer>(5);

    System.out.println("El recíproco de iOb es " +
                       iOb.reciprocal());
    System.out.println("El componente fraccionario de iOb es " +
                       iOb.fraction());

    System.out.println();

    FunsNumeric<Double> dOb =
                      new FunsNumeric<Double>(5.25);
```

En este caso, el argumento de tipo debe ser **Number** o una subclase de **Number**.

Integer es correcto porque es una subclase de **Number**.

Double también es correcto.

```
    System.out.println("El recíproco de dOb es " +
                       dOb.reciprocal());
    System.out.println("El componente fraccionario de dOb es " +
                       dOb.fraction());


    // Esto no se compilará porque String no es una
    // subclase de Number.
//    FunsNumeric<String> cadOb = new FunsNumeric<String>("Error");
String es ilegal porque no es una subclase de Number.
  }
}
```

String es ilegal porque no es una subclase de **Number**.

Ésta es la salida:

```
El recíproco de iOb es 0.2
El componente fraccionario de iOb es 0.0

El recíproco de dOb es 0.19047619047619047
El componente fraccionario de dOb es 0.25
```

Observe cómo **FunsNumeric** se encuentra declarado ahora por esta línea:

```
class FunsNumeric<T extends Number> {
```

Debido a que el tipo **T** está limitado ahora por **Number**, el compilador de Java sabe que todos los objetos de tipo **T** pueden llamar a **doubleValue()** porque es un método declarado por **Number**. Ésta constituye, en sí, una ventaja importante. Sin embargo, como bono adicional, el límite de **T** también evita la creación de objetos de **FunsNumeric** que no sean numéricos. Por ejemplo, si trata de eliminar los comentarios de las líneas al final del programa, y luego intenta volver a compilar, recibirá errores en tiempo de compilación, porque **String** no es una subclase de **Number**.

Los tipos limitados son especialmente útiles cuando necesita asegurar que un parámetro de tipo sea compatible con otro. Por ejemplo, considere la siguiente clase llamada **Par**, que almacena dos objetos que deben ser compatibles entre sí.

```
class Par<T, V extends T>
  T primero
  V segundo

  Par(T a, V b) {
    primero = a;
    segundo = b;
  }

  // ...
}
```

Aquí, **V** debe ser del mismo tipo que **T**, o una subclase de **T**.

Observe que **Par** usa dos parámetros de tipo, **T** y **V**, y que **V** extiende **T**. Esto significa que **V** será igual que **T** o una subclase de **T**. Esto asegura que los dos argumentos del constructor de **Par** sean objetos del mismo tipo o de tipos relacionados. Por ejemplo, las siguientes construcciones son válidas:

```
// Esto es correcto porque T y V son Integer.
Par<Integer, Integer> x = new Par<Integer, Integer>(1, 2);

// Esto es correcto porque Integer es una subclase de Number.
Par<Number, Integer> y = new Par<Number, Integer>(10.4, 12);
```

Sin embargo, lo siguiente no es válido:

```
// Esto causa un error porque String no es
// una subclase de Number
Par<Number, String> < = new Par<Number, String>(10.4, "12");
```

En este caso, **String** no es una subclase de **Number**, lo cual viola el límite especificado por **Par**.

Comprobación de avance

1. La palabra clave _____ especifica un límite para un argumento de tipo.
2. ¿Cómo declara un tipo genérico **T** que debe ser una subclase de **Thread**?
3. Dado

   ```
   class X<T, V extends T> {
   ```

   ¿la siguiente declaración es correcta?

   ```
   X<Integer, Double> x = new X<Integer, Double>(10, 1.1);
   ```

HABILIDAD FUNDAMENTAL 13.4

Uso de argumentos comodín

Aunque la seguridad de tipo es útil, puede llegar a estorbar en construcciones perfectamente aceptables. Por ejemplo, dada la clase **FunsNumeric** que se mostró al final de la sección anterior, suponga que quiere agregar un método llamado **absIgual()** que regrese **true** si dos objetos de

1. **extends**
2. `T extends Thread`
3. No, porque **Double** no es una subclase de **Integer**.

FunsNumeric contienen los números cuyos valores absolutos son iguales. Más aún, quiere que este método funcione apropiadamente sin importar el tipo de número que cada objeto contiene. Por ejemplo, si un objeto contiene el valor **Double** 1.25 y el otro objeto contiene el valor **Float** –1.25, entonces **absIgual( )** regresaría true. Una manera de implementar **absIgual( )** consiste en pasarlo a un argumento de **FunsNumeric** y luego comparar el valor absoluto del archivo contra el valor absoluto del objeto que invoca, regresando true sólo si los valores son iguales. Por ejemplo, tal vez quiera llamar a **absIgual( )** como se muestra aquí:

```
FunsNumeric<Double> dOb = new FunsNumeric<Double>(1,25);
FunsNumeric<Float> fOb = new FunsNumeric<Float>(-1.25);

if(dOb.absIgual(fOb))
  System.out.println("Los valores absolutos son iguales.");
else
  System.out.println("Los valores absolutos son diferentes.");
```

Al principio, la creación de **absIgual( )** parece ser un problema fácil. Por desgracia, los problemas empiezan en cuanto trata de declarar un parámetro de tipo **FunsNumeric**. ¿Cuál tipo debe especificar para el parámetro de tipo de **FunsNumeric**? Al principio, podría pensar en una solución como ésta, en la que **T** se usa como parámetro de tipo:

```
// ¡Esto no funciona!
// Determina si los valores absolutos de dos objetos son iguales
boolean absIgual(FunsNumeric<T> ob) {
  if(Math.abs(num.doubleValue()) ==
      Math.abs(ob.num.doubleValue()) return true;

  return false;
}
```

En este caso, se usa el método estándar **Math.abs( )** para obtener el valor absoluto de cada número y luego se comparan los valores. El problema con este intento es que sólo funcionará con otros objetos de **FunsNumeric** cuyo tipo sea el mismo que el del objeto que invoca. Por ejemplo, si el objeto que invoca es de tipo **FunsNumeric<Integer>**, entonces el parámetro **ob** debe ser también de este mismo tipo. No puede emplearse, por ejemplo, para comparar un objeto de tipo **FunsNumeric<Double>**. Por lo tanto, este método no arroja una solución general (es decir, genérica).

Para crear un método **absIgual( )** genérico, debe usar otra función de los genéricos de Java: el *argumento comodín*. El argumento comodín está especificado por **?** y representa un tipo desconocido. He aquí una manera de escribir el método **absIgual( )** empleando un comodín:

```
// Determina si los valores absolutos de dos
objetos son iguales // boolean absIgual(FunsNumeric<?> ob) {
  if(Math.abs(num.doubleValue()) ==
```

← Observe el comodín.

```
      Math.abs(ob.num.doubleValue()) return true;

  return false;
}
```

Aquí, **FunsNumeric<?>** coincide con cualquier objeto de **FunsNumeric**, permitiendo con ello que cualquier par de objetos de **FunsNumeric** tenga comparados sus valores absolutos. El siguiente programa lo demuestra:

```
// Uso de un comodín.
class FunsNumeric<T extends Number> {
  T num;

  // Pasa al constructor una referencia a
  // un objeto numérico.
  FunsNumeric(T n) {
    num = n;
  }

  // Regresa el recíproco.
  double reciprocal() {
    return 1 / num.doubleValue();
  }

  // Regresa el componente fraccionario.
  double fraction() {
    return num.doubleValue() - num.intValue();
  }

  // Determina si los valores absolutos de dos
  // objetos son iguales.
  boolean absIgual(FunsNumeric<?> ob) {
    if(Math.abs(num.doubleValue()) ==
         Math.abs(ob.num.doubleValue())) return true;

    return false;
  }

  // ...
}

// Demuestra un comodín.
class ComodínDemo {
  public static void main(String args[]) {

    FunsNumeric<Integer> iOb =
                      new FunsNumeric<Integer>(6);
```

```
    FunsNumeric<Double> dOb =
                      new FunsNumeric<Double>(-6.0);

    FunsNumeric<Long> lOb =
                      new FunsNumeric<Long>(5L);

    System.out.println("Probando iOb y dOb.");
    if(iOb.absIgual(dOb))
      System.out.println("Los valores absolutos son iguales.");
    else
      System.out.println("Los valores absolutos son diferentes.");

    System.out.println();

    System.out.println("Probando iOb y lOb.");
    if(iOb.absIgual(lOb))
      System.out.println("Los valores absolutos son iguales.");
    else
      System.out.println("Los valores absolutos son diferentes.");

  }
}
```

En esta llamada, el tipo del comodín coincide con **Double**.

En esta llamada, el comodín coincide con **Long**.

Ésta es la salida:

```
Probando iOb y dOb.
Los valores absolutos son iguales.

Probando iOb y lOb.
Los valores absolutos son diferentes.
```

En el programa, note estas dos llamadas a **absIgual()**:

```
if(iOb.absIgual(dOb))

if(iOb.absIgual(lOb))
```

En la primera llamada, **iOb** es un objeto de tipo **FunsNumeric<Integer>** y **dOb** es un objeto de tipo **FunsNumeric<Double>**; sin embargo, mediante el uso de un comodín es posible para **iOb** pasar **dOb** en la llamada a **absIgual()**. Lo mismo se aplica a la segunda llamada, en la cual se pasa un objeto de tipo **FunsNumeric<Long>**.

Un último punto: es importante comprender que el comodín no afecta al tipo de objetos de **FunsNumeric** que se puede crear. Esto está gobernado por la cláusula **extends** en la declaración **FunsNumeric**. El comodín simplemente coincide con cualquier objeto *válido* de **FunsNumeric**.

HABILIDAD FUNDAMENTAL 13.5

Comodines limitados

Es posible limitar los argumentos de comodín de manera muy parecida a cómo se limita un parámetro de tipo. Un comodín limitado resulta especialmente importante cuando usted crea un método que está diseñado para operar sólo sobre objetos que son subclases de una superclase específica. Para comprender el porqué, revisemos un ejemplo simple. Considere el siguiente conjunto de clases:

```
class A {
  // ...
}

class B extends A {
  // ...
}

class C extends A {
  // ...
}

// Observe que D NO extiende A.
class D {
  // ...
}
```

Aquí, la clase **A** está extendida por las clases **B** y **C**, pero no por **D**.

A continuación, considere la siguiente clase genérica, la cual es muy simple:

```
// Una clase genérica simple.
class Gen<T> {
  T ob;

  Gen(T o) {
    ob = o;
  }
}
```

Gen toma un parámetro de tipo que especifica el tipo de **Object** almacenado en **ob**. Debido a que **T** no está limitado, el tipo de **T** carece de restricción alguna, es decir, **T** puede ser de cualquier tipo de clase.

Ahora suponga que desea crear un método que tome como argumento cualquier tipo de objeto de **Gen**, siempre y cuando su parámetro de tipo sea **A** o una subclase de **A**. En otras palabras, desea crear un método que sólo opere en objetos de **Gen**<*tipo*>, donde *tipo* es **A** o una subclase de **A**. Para

lograrlo, debe usar un comodín limitado. Por ejemplo, he aquí un método llamado **prueba()** que sólo acepta como argumento objetos de **Gen** cuyo parámetro de tipo es **A** o una subclase de **A**:

```
// Aquí, el ? coincidirá con A o cualquier tipo de clase
// que extienda A
static void prueba(Gen<? extends A> o) {
  // ...
}
```

La siguiente clase demuestra los tipos de objetos de **Gen** que pueden pasarse a **prueba()**.

```
class UsoComodínNoLimitado {
  // Aquí, el ? coincidirá con A o con cualquier tipo de clase
  // que extienda A
  static void prueba(Gen<? extends A> o) {   <-- Uso de un comodín limitado.
    // ...
  }
  public static void main(String args[]) {
    A a = new A();
    B b = new B();
    C c = new C();
    D d = new D();

    Gen<A> w = new Gen<A>(a);
    Gen<B> w2 = new Gen<B>(b);
    Gen<C> w3 = new Gen<C>(c);
    Gen<D> w4 = new Gen<D>(d);

    // Estas llamadas a prueba() son correctas.
    prueba(w);
    prueba(w2);   <-- Éstas son legales porque w, w2 y w3 son subclases de A.
    prueba(w3);

    // No puede llamar a prueba() con w4 porque
    // no es un objeto de una clase que
    // herede A.
//    prueba(w4); // ¡Error!   <-- Esto es ilegal porque w4 no es una subclase de A.
  }
}
```

En **main()** se crean objetos de tipo **A**, **B**, **C** y **D**. Éstos son utilizados para crear cuatro objetos de **Gen**, uno para cada tipo. Por último, se realizan cuatro llamadas a **prueba()** la última llamada está

convertida en comentario. Las primeras tres llamadas son válidas porque **w**, **w2** y **w3** son objetos de **Gen** cuyo tipo es **A** o una subclase de **A**. Sin embargo, la última llamada a **prueba ()** es ilegal porque **w4** es un objeto de tipo **D**, que no se deriva de **A**. Por lo tanto, el comodín limitado en **prueba( )** no aceptará **w4** como argumento.

En general, para establecer el límite superior para un comodín, use el siguiente tipo de 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 ya que la clase que forma el límite superior (es decir, especificado por *superclase*) también se encuentra dentro de límites.

Asimismo, puede especificar un límite inferior para un comodín si agrega una cláusula **super** a una declaración de comodín. He aquí la forma general:

<? super *subclase*>

En este caso, sólo las clases que son superclases de una *subclase* son argumentos aceptables. Se trata de una cláusula exclusiva porque no coincidirá con la clase especificada por *subclase*.

Comprobación de avance

1. Para especificar un argumento comodín, se usa _____.
2. Un argumento comodín coincide con cualquier tipo de referencia. ¿Cierto o falso?
3. ¿Un comodín puede estar limitado?
4. En esta expresión, ¿con qué tipo de objetos puede coincidir el comodín?

```
void MiMet(XYZ<? extends Thread> sunprOb) { // ...
```

1. **?**
2. Cierto.
3. Sí.
4. El comodín puede coincidir con cualquier objeto que sea de tipo **Thread** o una subclase de **Thread**.

Pregunte al experto

P: **¿Puedo moldear una instancia de una clase genérica en otra?**

R: Sí, puede moldear una instancia de una clase genérica en otra, pero sólo si las dos son de otra manera compatibles y si sus argumentos de tipo son iguales. Por ejemplo, suponga una clase genérica llamada **Gen** que se declare del siguiente modo:

```
class Gen<T> { // ...
```

A continuación, suponga que se declara **x** como se muestra aquí:

```
Gen<Integer> x = new Gen<Integer>();
```

Entonces, este moldeo es legal:

```
(Gen<Integer>) x // legal
```

porque **x** es un instancia de **Gen<Integer>**. Pero este moldeo

```
(Gen<Long> x // ilegal
```

no es legal porque **x** no es una instancia de **Gen<Long>**.

HABILIDAD FUNDAMENTAL 13.6

Métodos genéricos

Como se ha mostrado en los ejemplos anteriores, los métodos dentro de una clase genérica pueden utilizar el parámetro de tipo de clase y son, por consiguiente, automáticamente genéricas en relación con el parámetro de tipo. Sin embargo, es posible declarar un método genérico que utilice uno o más parámetros propios de tipo. Más aún, es posible crear un método genérico que se encuentre dentro de una clase no genérica.

El siguiente programa declara una clase no genérica llamada **MétodoGenéricoDemo** y un método genérico estático dentro de la clase llamado **matricesIguales()**. Este método determina si dos matrices contienen los mismos elementos en el mismo orden. Pueden emplearse para comparar dos matrices, siempre y cuando las matrices sean del mismo tipo o de tipos compatibles.

```
// Demuestra un método genérico simple.
class MétodoGenéricoDemo {

  // Determina si los contenidos de dos matrices son iguales.
```

```
  static <T, V extends T> boolean matricesIguales(T[] x, V[] y) {
    // si la longitud de las matrices difiere, las matrices son diferentes.
    if(x.length != y.length) return false;

    for(int i=0; i < x.length; i++)
      if(x[i] != y[i]) return false; // las matrices difieren

    return true; // el contenido de las matrices es equivalente
  }

  public static void main(String args[]) {

    Integer nums[] = { 1, 2, 3, 4, 5 };
    Integer nums2[] = { 1, 2, 3, 4, 5 };
    Integer nums3[] = { 1, 2, 7, 4, 5 };
    Integer nums4[] = { 1, 2, 7, 4, 5, 6 };

    if(matricesIguales(nums, nums))
      System.out.println("nums es igual a nums");

    if(matricesIguales(nums, nums2))
      System.out.println("nums es igual a nums2");

    if(matricesIguales(nums, nums3))
      System.out.println("nums es igual a nums3");

    if(matricesIguales(nums, nums4))
      System.out.println("nums es igual a nums4");

    // Crea una matriz de Doubles
    Double dvals[] = { 1.1, 2.2, 3.3, 4.4, 5.5 };

    // Esto no se compilará porque nums y dvals
    // no son del mismo tipo.
//    if(matricesIguales(nums, dvals))
//      System.out.println("nums es igual a dvals");
  }
}
```

Un método genérico.

Los argumentos de tipo para **T** y **V** se encuentran determinados implícitamente cuando se llama al método.

Ésta es la salida de este programa:

```
nums es igual a nums
nums es igual a nums2
```

Examinemos **matricesIguales( )** de cerca. En primer lugar, observe cómo se declara en esta línea:

```
static <T, V extends T> boolean matricesIguales(T[] x, V[] y) {
```

Los parámetros de tipo se declaran *antes* del tipo de regreso del método. En segundo lugar, observe que el tipo *V* tiene el límite superior en **T**. Por lo tanto, **V** debe ser igual que el tipo **T** o una subclase de **T**. Esta relación impone que **matricesIguales( )** sea llamado sólo mediante argumentos que sean compatibles entre sí. Además, note que **matricesIguales( )** es estático, lo que permite que sea llamado de manera independiente a cualquier objeto. Sin embargo, observe que los métodos genéricos pueden ser estáticos o no estáticos, pues no existe ninguna restricción al respecto.

Ahora, observe cómo se llama a **matricesIguales( )** dentro de **main( )** mediante el uso de la sintaxis de llamada normal, sin la necesidad de especificar argumentos de tipo. Esto se debe a que los tipos de argumentos son discernidos automáticamente y a que los tipos de **T** y **V** se ajustan de manera correspondiente. Por ejemplo, en la primera llamada:

```
if(matricesIguales(nums, nums))
```

el tipo de base del primer argumento es **Integer**, el cual hace que **Integer** se sustituya con **T**. El tipo de base del segundo argumento también es **Integer**, el cual también hace que **Integer** sea un sustituto de **V**. Así, la llamada a **matricesIguales ( )** es legal y las dos matrices pueden compararse.

Ahora observe aquello que se aparta del código al convertirlo en comentario:

```
//    if(matricesIguales(nums, dvals))
//      System.out.println("nums es igual a dvals");
```

Si elimina los comentarios y trata de compilar el programa, recibirá un error. La razón es que el parámetro de tipo **V** está limitado por **T** en la cláusula **extends** de 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 **Integer**, lo que convierte **T** en **Integer**; sin embargo, el segundo argumento es de tipo **Double**, el cual no es una subclase de **Integer**. Esto hace que la llamada a **matricesIguales( )** sea ilegal y genere como resultado un error de falta de coincidencia de tipo en tiempo de compilación.

Es posible generalizar la sintaxis que se emplea para crear **matricesIguales( )**. He aquí la sintaxis de un método genérico:

<lista-params-tipo> tipo-reg nombre-método(*lista-parámetros*) { // ...

En todos los casos, *lista-params-tipo* es una lista de parámetros de tipo separados por comas. Observe que para un método genérico, la lista de parámetros de tipo antecede al tipo de regreso.

HABILIDAD FUNDAMENTAL 13.7

Constructores genéricos

Un constructor puede ser genérico, aunque su clase no lo sea. Por ejemplo, en el siguiente programa la clase **Sumatoria** no es genérica, pero su constructor sí lo es.

```
// Uso de un constructor genérico.
class Sumatoria {
  private int sum;

  <T extends Number> Sumatoria(T arg) {   <-- Un constructor genérico.
    sum = 0;

    for(int i=0; i <= arg.intValue(); i++)
      sum += i;
  }

  int obtenerSum() {
    return sum;
  }
}

class ConsGenDemo {
  public static void main(String args[]) {
    Sumatoria ob = new Sumatoria(4.0);

    System.out.println("La sumatoria de 4.0 es " +
                       ob.obtenerSum());
  }
}
```

La clase **Sumatoria** calcula y encapsula la sumatoria del valor numérico que es pasado a su constructor. Recuerde que la sumatoria de *N* es la suma de todos los números enteros entre 0 y *N*. Debido a que **Sumatoria( )** especifica un parámetro de tipo que está limitado por **Number**, un objeto de **Sumatoria** puede construirse usando cualquier tipo numérico, incluidos **Integer**, **Float** o **Double**. No importa el tipo numérico que se use, su valor se convierte a **Integer** al llamar a **intValue( )** y se calcula la sumatoria. De modo que no es necesario que la clase **Sumatoria** sea genérica, sólo se requiere un constructor genérico.

Comprobación de avance

1. ¿Un método o constructor puede ser genérico aunque su clase no lo sea?
2. Muestre cómo declarar un método genérico llamado **MiMet( )** que toma un argumento de tipo genérico. Haga que éste regrese un argumento de dicho tipo genérico.

HABILIDAD FUNDAMENTAL 13.8

Interfaces genéricas

Además de las clases y los métodos genéricos, puede también tener interfaces genéricas. Las interfaces genéricas se especifican de la misma manera que las clases genéricas. He aquí un ejemplo que crea una interfaz llamada **Contenedor**, que puede implementarse con clases que almacenan uno o más valores; y declara un método llamado **contiene( )** que determina si un valor especificado está contenido por el objeto que convoca.

```
// Un ejemplo de interfaz genérica.

// Una interfaz de contenedor genérico.
// Esta interfaz implica que una clase de
// implementación contiene uno o más valores.
interface Contenedor<T> {
  // El método contiene() pruebas para saber si
  // un elemento específico se encuentra dentro de
  // un objeto que implementa Contenedor.
  boolean contiene(T o);
}

// Implementa Contenedor usando una matriz para
// contener los valores.
class MiClase<T> implements Contenedor<T> {
  T[] refMatriz;

  MiClase(T[] o) {
    refMatriz = o;
  }
```

Una interfaz genérica.

Cualquier clase que implementa una interfaz genérica debe ser genérica en sí misma.

1. Sí.
2. `<T> T MiMet(T o)`

```
  // Implementa contiene.
  public boolean contiene(T o) {
    for(T x : refMatriz)
      if(x.equals(o)) return true;
    return false;
  }
}

class IFGenDemo {
  public static void main(String args[]) {
    Integer x[] = { 1, 2, 3 };

    MiClase<Integer> ob = new MiClase<Integer>(x);

    if(ob.contiene(2))
      System.out.println("2 es un ob");
    else
      System.out.println("2 NO es un ob");

    if(ob.contiene(5))
      System.out.println("5 es un ob");
    else
      System.out.println("5 NO es un ob");

    // Lo siguiente es ilegal porque ob
    // es un contenedor de Integer y 9.25 es
    // un valor Double.
//    if(ob.contiene(9.25)) // ¡Ilegal!
//      System.out.println("9.25 es un ob");

  }
}
```

Ésta es la salida:

```
2 es un ob
5 NO es un ob
```

Aunque resulta fácil comprender casi todos los aspectos de este programa, es necesario aclarar un par de puntos clave. En primer lugar, observe que **Contenedor** se declara de esta manera:

```
interface Contenedor<T> {
```

En general, una interfaz genérica se declara de la misma manera que una clase genérica. En este caso, el parámetro de tipo **T** especifica el tipo de objetos que contiene.

A continuación, **Contenedor** está implementada por **MiClase**. Observe la declaración de **MiClase** que se muestra aquí.

```
class MiClase<T> implements Contenedor<T> {
```

En general, si una clase implementa una interfaz genérica, entonces esa clase debe ser también genérica, al menos en la medida en que toma un parámetro de tipo que se pasa a la interfaz. Por ejemplo, el siguiente intento de declarar **MiClase** constituye un error:

```
class MiClase implements Contenedor<T> { // ¡Incorrecto!
```

Esta declaración es incorrecta porque **MiClase** no declara un parámetro de tipo, lo que significa que no hay manera de pasar uno a **Contenedor**. En este caso, simplemente se desconoce al identificador **T** y el compilador reporta un error. Por supuesto, si una clase implementa un *tipo específico* de interfaz genérica, como se muestra aquí:

```
class MiClase implements Contenedor<Double> { // Correcto
```

entonces la clase que implementa no necesita ser genérica.

Como es de esperarse, es posible que el parámetro o los parámetros de tipo especificados por una interfaz genérica estén limitados. Esto le permite limitar el tipo de datos a los que es posible implementar la interfaz. Por ejemplo, si quisiera limitar **Contenedor** a tipos numéricos, entonces podría declararlo así:

```
interface Contenedor<T extends Number> {
```

Ahora, cualquier clase que implemente debe pasar a **Contenedor** un argumento de tipo que también tenga el mismo límite. Por ejemplo, **MiClase** debe declararse aquí:

```
class MiClase<T extends Number> implements Contenedor<T> {
```

Preste especial atención a la manera en que se declara el parámetro de tipo **T** en **MiClase** y luego se pasa a **Contenedor**. Debido a que ahora **Contenedor** requiere un tipo que extienda **Number**, la clase que implementa (en este caso, **MiClase**) debe especificar el mismo límite. Más aún, una vez que se ha establecido este límite, no hay necesidad de especificarlo de nuevo en la cláusula **implements**. En realidad resultaría incorrecto hacerlo. Por ejemplo, esta declaración es incorrecta y no se compilará:

```
// ¡Esto es incorrecto!
class MiClase<T extends Number>
  implements Contenedor<T extends Number> { // ¡Incorrecto!
```

Una vez que se ha establecido el parámetro de tipo, simplemente se pasa a la interfaz sin una modificación adicional.

He aquí la sintaxis generalizada de una interfaz genérica:

interface *nombre-interfaz<lista-params-tipo>* { // ...

Aquí, *lista-params-tipo* es una lista de parámetros de tipo separados por comas. Cuando se implementa una interfaz genérica, se debe especificar el argumento de tipo como se muestra aquí.

class *nombre-clase<lista-params-tipo>*
  implements *nombre-interfaz<lista-params-tipo>* {

Proyecto 13.1 Cree una cola genérica

CIGen.java
CExc.java
ColaGen.java
CGenDemo.java

Una de las ventajas más importantes que los elementos genéricos proporcionan a la programación es la capacidad de construir un código confiable y reciclable. Como se mencionó al principio de este módulo, muchos algoritmos son iguales, sin importar el tipo de datos que se utilice. Por ejemplo, una cola funciona de la misma manera sin importar que sean enteros, cadenas u objetos **File**. En lugar de crear una clase de cola separada para cada tipo de objeto, puede crear una sola solución genérica que pueda emplearse con cualquier tipo de objeto. Por lo tanto, el ciclo de desarrollo de diseño, código, prueba y depuración sólo ocurre una vez cuando crea una solución genérica (no de manera repetida, es decir, cada vez que se requiere una cola para un nuevo tipo de datos).

En este proyecto, adaptaremos el ejemplo de cola que se ha venido desarrollando desde el proyecto 5.2, para hacerla genérica. Este proyecto representa la evolución final de la cola e incluye una interfaz genérica que define las operaciones de cola, dos clases de excepción y una implementación de la cola: una cola de tamaño fijo. Por supuesto, usted puede experimentar con otros tipos de colas genéricas, como una cola dinámica genérica o una circular genérica. Tan sólo siga el ejemplo que se muestra aquí.

Este proyecto organiza también el código de cola en un conjunto de archivos separados: uno para la interfaz, uno para las excepciones de la cola, uno para la implementación de cola fija, y uno para el programa que lo demuestra. Esta organización refleja la manera en que este proyecto se organizaría normalmente en la realidad.

Paso a paso

1. El primer paso en la creación de una cola genérica es crear una interfaz genérica que describa las dos operaciones de la cola: colocar y obtener. La versión genérica de la interfaz de cola es **CIGen**, la cual se muestra a continuación. Coloque esta interfaz en un archivo llamado **CIGen.java**.

```
// Una interfaz de cola genérica.
public interface CIGen<T> {
  // Coloca un elemento en la cola.
  void put(T ch) throws QueueFullException;
```

```
  // Obtiene un elemento de la cola.
  T get() throws QueueEmptyException;
}
```

Observe que el tipo de datos almacenados en la cola se especifica mediante el parámetro de tipo genérico **T**.

2. A continuación, cree el archivo **CExc.java** y agréguele las siguientes dos cláusulas de excepción:

```
// Una excepción para errores de cola llena.
class QueueFullException extends Exception {
  int dimen;

  QueueFullException(int d) { dimen = d; }

  public String toString() {
   return "\nLa cola se ha llenado. El tamaño máximo es " +
          dimen;
  }
}

// Una excepción para errores de cola vacía.
class QueueEmptyException extends Exception {

  public String toString() {
   return "\nLa cola está vacía.";
  }
}
```

Estas clases encapsulan los dos errores de la cola: llena o vacía. No son clases genéricas puesto que son iguales, sin importar el tipo de datos que se almacene en una cola.

3. Ahora, cree un archivo llamado **ColaGen.java**. Coloque en él el siguiente código, el cual implementa una cola de tamaño fijo:

```
// Una clase genérica de tamaño fijo.
class ColaGen<T> implements CIGen<T> {
  private T q[]; // esta matriz contiene la cola
  private int colocarlug, obtenerlug; // los índices colocar y poner

  // Construye una cola vacía con la matriz dada.
  public ColaGen(T[] aRef) {
    q = aRef;
    colocarlug = obtenerlug = 0;
  }

  // Coloca un elemento en la cola.
  public void put(T obj)
    throws QueueFullException {
```

(continúa)

```
    if(colocarlug==q.length-1)
      throw new QueueFullException(q.length-1);

    colocarlug++;
    q[colocarlug] = obj;
  }

  // Obtiene un carácter de la cola.
  public T get()
    throws QueueEmptyException {

    if(obtenerlug == colocarlug)
      throw new QueueEmptyException();

    obtenerlug++;
    return q[obtenerlug];
  }
}
```

ColaGen es una clase genérica con parámetro de tipo **T**, que especifica el tipo de datos almacenados en la cola. Observe que **T** también se pasa a la interfaz **CIGen**.

Observe que el constructor **ColaGen** pasa una referencia a una matriz que se utilizará para contener la cola. Por consiguiente, para construir una **ColaGen**, primero deberá crear una matriz cuyo tipo sea compatible con el objeto que estará almacenando en la cola y cuyo tamaño sea lo suficientemente largo para almacenar el número de objetos que se colocará en la cola. A medida que se escribe el código, la matriz debe ser mucho más larga que el número de elementos almacenados ya que la primera ubicación queda sin uso.

Por ejemplo, la siguiente secuencia muestra cómo crear una cola que contiene cadenas:

```
String cadMatriz[] = new String[10];
ColaGen<String> cCad = new ColaGen<String>(cadMatriz);
```

4. Cree un archivo llamado **CGenDemo.java** y coloque en él el siguiente código. Este programa demuestra la cola genérica.

```
/*
    Proyecto 13.1

    Demuestra una clase de cola genérica.
*/
class CGenDemo {
  public static void main(String args[]) {
    // Crea una cola de enteros.
    Integer iAlmacena[] = new Integer[10];
    ColaGen<Integer> c = new ColaGen<Integer>(iAlmacena);
```

```
Integer iVal;

System.out.println("Demuestra una cola de enteros.");
try {
  for(int i=0; i < 5; i++) {
    System.out.println("Sumando " + i + " a la c.");
    c.put(i); // agrega un valor entero a c

  }
}
catch (QueueFullException exc) {
  System.out.println(exc);
}
System.out.println();

try {
  for(int i=0; i < 5; i++) {
    System.out.print("Obteniendo el siguiente entero de c: ");
    iVal = c.get();
    System.out.println(iVal);
  }
}
catch (QueueEmptyException exc) {
  System.out.println(exc);
}

System.out.println();

// Crea una cola de Double.
Double dAlmacena[] = new Double[10];
ColaGen<Double> c2 = new ColaGen<Double>(dAlmacena);

Double dVal;

System.out.println("Demuestra una cola de Doubles.");
try {
  for(int i=0; i < 5; i++) {
    System.out.println("Sumando " + (double)i/2 +
                       " a la c2.");
    c2.put((double)i/2); // Suma valor double a c2
  }
}
catch (QueueFullException exc) {
  System.out.println(exc);
}
```

(continúa)

```
    System.out.println();

    try {
      for(int i=0; i < 5; i++) {
        System.out.print("Obtención del siguiente Double de c2: ");
        dVal = c2.get();
        System.out.println(dVal);
      }
    }
    catch (QueueEmptyException exc) {
      System.out.println(exc);
    }
  }
}
```

5. Compile el programa y ejecútelo. Verá la salida que a continuación se muestra:

```
Demuestra una cola de enteros.
Sumando 0 a la c.
Sumando 1 a la c.
Sumando 2 a la c.
Sumando 3 a la c.
Sumando 4 a la c.

Obteniendo el siguiente entero de c: 0
Obteniendo el siguiente entero de c: 1
Obteniendo el siguiente entero de c: 2
Obteniendo el siguiente entero de c: 3
Obteniendo el siguiente entero de c: 4

Demuestra una cola de Doubles.
Sumando 0.0 a la c2.
Sumando 0.5 a la c2.
Sumando 1.0 a la c2.
Sumando 1.5 a la c2.
Sumando 2.0 a la c2.

Obteniendo el siguiente Double de c2: 0.0
Obteniendo el siguiente Double de c2: 0.5
Obteniendo el siguiente Double de c2: 1.0
Obteniendo el siguiente Double de c2: 1.5
Obteniendo el siguiente Double de c2: 2.0
```

6. Por su cuenta, trate de convertir las clases **ColaCircular** y **ColaDin** del proyecto 8.1 en clases genéricas.

HABILIDAD FUNDAMENTAL 13.9

Tipos brutos y código heredado

Debido a que los elementos genéricos constituyen una nueva función, era necesario que Java proporcionara algún camino de transición para el código antiguo, previo a los elementos genéricos. En la actualidad, aún hay millones y millones de líneas de código heredado que es previo a los elementos genéricos y que seguramente sigue siendo funcional y compatible con los elementos genéricos. Esto significa que el código anterior puede funcionar con elementos genéricos en tanto que el código con elementos genéricos debe tener la capacidad de trabajar con el código previo.

Para manejar la transición a los elementos genéricos, Java permite que una clase genérica se use sin argumentos de tipo, lo cual crea un *tipo bruto* para la clase. Este tipo bruto es compatible con código heredado, el cual no conoce los elementos genéricos. La principal desventaja de usar el tipo bruto es que se pierde la seguridad de tipo de los elementos genéricos.

He aquí un ejemplo que muestra un tipo bruto en acción.

```
// Demuestra un tipo 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;
  }

  // Regresa ob.
  T obtenerob() {
    return ob;
  }
}

// Demuestra un tipo bruto.
class BrutoDemo {
  public static void main(String args[]) {

    // Crea un objeto de Gen para enteros.
    Gen<Integer> iOb = new Gen<Integer>(88);

    // Crea un objeto de Gen para cadenas.
    Gen<String> cadOb = new Gen<String>("Prueba de genéricos");

    // Crea un tipo bruto de un objeto de Gen y da
    // un valor Double.
    Gen bruto = new Gen(new Double(98.6)); ← Cuando no se proporciona un argumento de tipo, se crea un tipo bruto.
```

```
        // Aquí el moldeo es necesario porque el tipo resulta desconocido.
        double d = (Double) bruto.obtenerob();
        System.out.println("valor: " + d);

        // El uso de un tipo bruto puede llevar a excepciones
        // en tiempo de ejecución. He aquí algunos ejemplos.

        // El siguiente moldeo causa un error en tiempo de ejecución.
    //    int i = (Integer) bruto.obtenerob(); // error en tiempo de ejecución

        // Esta asignación sobrescribe la seguridad de tipo.
        cadOb = bruto; // Correcto, pero posiblemente equivocado
//    String cad = cadOb.obtenerob(); // error en tiempo de ejecución

    // Esta asignación también sobrescribe la seguridad de tipo.
    bruto = iOb; // Correcto, pero posiblemente equivocado
//    d = (Double) bruto.obtenerob(); // error en tiempo de ejecución
  }
}
```

Los tipos brutos sobrescriben la seguridad de tipo.

Este programa contiene varios aspectos interesantes. En primer lugar, un tipo bruto de la clase genérica **Gen** se crea mediante la siguiente declaración:

```
Gen bruto = new Gen(new Double(98.6));
```

Observe que no se especifican argumentos de tipo. En esencia, esto crea un objeto de **Gen** cuyo tipo **T** se reemplaza con **Object**.

Un tipo bruto no cuenta con seguridad de tipo. Por lo tanto, a una variable de tipo bruto se le puede asignar una referencia a cualquier tipo de objeto de **Gen**. Lo contrario también se permite, en cuyo caso a una variable de un tipo específico de **Gen** se le puede asignar una referencia a un objeto bruto de **Gen**. Sin embargo, ambas operaciones resultan potencialmente inseguras puesto que es posible darle la vuelta al tipo de mecanismo de comprobación de elementos genéricos.

Esta falta de seguridad de tipo se ilustra mediante las líneas convertidas en comentarios al final del programa. Examinemos cada caso. En primer lugar, considere la siguiente situación:

```
//    int i = (Integer) bruto.obtenerob(); // error en tiempo de ejecución
```

En esta declaración, el valor de **ob** dentro de **bruto** contiene un valor **Double**, no uno **Integer**. Sin embargo, esto no puede detectarse en tiempo de compilación porque el tipo de **bruto** es desconocido. De ahí que esta instrucción falle en tiempo de ejecución.

La siguiente secuencia asigna a una **cadOb** (una referencia de tipo **Gen<String>**) una referencia a un objeto bruto de **Gen**:

```
cadOb = bruto; // Correcto, pero posiblemente equivocado
//    String cad = cadOb.obtenerob(); // error en tiempo de ejecución
```

La asignación por sí misma resulta sintácticamente correcta, pero cuestionable. Como **cadOb** es de tipo **Gen<String>**, se supone que contiene una cadena. Sin embargo, después de la asignación, el objeto al que hace referencia **cadOb** contiene un **Double**. Por tanto, cuando se hace el intento, en tiempo de ejecución, de asignar el contenido de **cadOb** a **cad**, se genera un error en tiempo de ejecución porque **cadOb** contiene ahora un **Double**. En consecuencia, la asignación de una referencia bruta a una referencia genérica pasa por alto el mecanismo de seguridad de tipo.

La siguiente secuencia invierte el caso anterior.

```
bruto = iOb; // Correcto, pero potencialmente equivocado
//    d = (Double) bruto.obtenerob(); // error en tiempo de ejecución
```

Aquí, se asigna una referencia genérica a la variable de referencia bruta. Aunque esto es sintácticamente correcto, puede provocar problemas, tal y como lo ilustra la segunda línea. En este caso, **bruto** hace referencia ahora a un objeto que contiene un objeto **Integer**; sin embargo, el moldeo supone que contiene un **Double**. Este error no puede evitarse en tiempo de compilación. En cambio, causa un error en tiempo de ejecución.

Debido a la posibilidad inherente de peligro de los tipos brutos, **javac** despliega *avisos de no comprobación* cuando se usa un tipo bruto de una manera que podría poner en peligro la seguridad de tipo. En el programa anterior, estas líneas generan avisos:

```
Gen bruto = new Gen(new Double(98.6));

cadOb = bruto; // Correcto, pero potencialmente equivocado
```

En la primera línea, la llamada al constructor de **gen** sin un argumento de tipo es la que causa el aviso. En la segunda línea, la asignación de una referencia bruta a una variable genérica es la que genera los avisos.

Al principio, es posible que piense que esta línea debe generar también un aviso de no comprobación, pero no es así.

```
bruto = iOb; // Correcto, pero potencialmente equivocado
```

El compilador no emite avisos porque la asignación no causa ninguna pérdida *adicional* de la seguridad de tipo que no haya ocurrido antes cuando se creó **bruto**.

Una consideración final: debe limitar el uso de tipos brutos a los casos en que debe mezclar código heredado con código genérico más reciente. Los tipos brutos constituyen simplemente una función de transición y no algo que deba emplearse para un nuevo código.

Comprobación de avance

1. Si una interfaz genérica se encuentra implementada por una clase, dicha clase debe ser también genérica. ¿Cierto o falso?
2. No es posible limitar un parámetro de tipo en una interfaz genérica. ¿Cierto o falso?
3. Dado

```
class XYZ<T> { // ...
```

demuestre cómo declarar un objeto llamado **ob** el cual es un tipo bruto de **XYZ**.

HABILIDAD FUNDAMENTAL 13.10

Borrado

Por lo general, no es necesario que el programador conozca los detalles sobre la manera en que el compilador de Java transforma su código fuente en código de objeto. Sin embargo, en el caso de los elementos genéricos, es importante que se tenga cierta comprensión general del proceso, ya que éste explica la razón de que las funciones genéricas funcionen como lo hacen (y la razón de que su comportamiento resulte un poco sorprendente en ocasiones). Por este motivo, se sugiere llevar a cabo un breve análisis sobre la manera en que se implantan los elementos genéricos en Java.

Una restricción importante que determinó la forma en que se agregaron los elementos genéricos a Java fue la necesidad de compatibilidad con versiones anteriores de Java. En otras palabras, el código genérico tiene que ser compatible con el código anterior, el cual no es genérico. Así, cualquier cambio a la sintaxis del lenguaje Java, o a la JVM, debía evitar el rompimiento del código anterior. Java implementa los elementos genéricos mientras satisface esta restricción mediante el uso del *borrado*.

En general, he aquí cómo funciona el borrado: cuando su código de Java se compila, se elimina (borra) toda la información de tipo genérico, lo que significa el reemplazo de parámetros de tipo con su tipo límite, el cual es **Object** si no hay un límite explícito; y después aplicar los moldeos apropiados (como los argumentos de tipo lo determinan) para mantener la compatibilidad de tipo con los tipos especificados por los argumentos de tipo. El compilador impone también esta compatibilidad de tipo. Este acercamiento a los elementos genéricos significa que no existen parámetros de tipo en tiempo de ejecución, pues son simplemente un mecanismo de código fuente.

Para comprender mejor el funcionamiento del borrado, considere las siguientes dos clases:

```
// Aquí, T está limitado por Object como opción predeterminada.
class Gen<T> {
  T ob; // Aquí, T será reemplazada por Object
```

1. Cierto.
2. Falso.
3. `XYZ ob = new XYZ();`

```
  Gen(T o) {
    ob = o;
  }

  // Regresa ob.
  T obtenerob() {
    return ob;
  }
}

// Aquí, T está limitado por String.
class CadGen<T extends String> {
  T cad; // aquí, T será reemplazado por String

  CadGen(T o) {
    cad = o;
  }

  T obtenercad() { return cad; }
}
```

Después de que estas dos clases son compiladas, la **T** en **Gen** se reemplazará con **Object** y la **T** en **CadGen** se reemplazará con **String**. Dentro del código de **Gen** y **CadGen**, los moldeos implícitos se emplean para borrar el tipo apropiado. Por ejemplo, esta secuencia

```
Gen<Integer> iOb = new Gen<Integer>(99);

int x = iOb.obtenerob();
```

se compilaría como si estuviera escrita de la siguiente forma:

```
Gen iOb = new Gen(99):
Int x = (Integer) iOb.obtenerob():
```

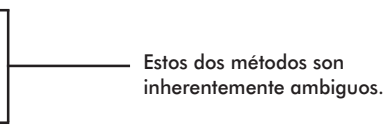
HABILIDAD FUNDAMENTAL 13.11

Errores de ambigüedad

La inclusión de los elementos genéricos plantea un nuevo tipo de error ante el cual debe estar prevenido: la *ambigüedad*. Los errores de ambigüedad se presentan cuando el borrado hace que dos

declaraciones genéricas aparentemente distintas se resuelvan en el mismo tipo borrado, lo que origina un conflicto. He aquí un ejemplo que incluye la sobrecarga de métodos:

```
// La ambigüedad provocada por el borrado en
// métodos sobrecargados.
class MiClaseGen<T, V> {
  T ob1;
  V ob2;

  // ...

  // Estos dos métodos sobrecargados son ambiguos
  // y no se compilarán.
  void set(T o) {
    ob1 = o;
  }

  void set(V o) {
    ob2 = o;
  }
}
```



Observe que **MiClaseGen** declara dos tipos genéricos: **T** y **V**. Dentro de **MiClaseGen**, se hace un intento de sobrecargar **set( )** con base en parámetros de tipo **T** y **V**. Esto parece razonable porque, al parecer, **T** y **V** son tipos distintos. Sin embargo, hay aquí dos problemas de ambigüedad:

En primer lugar, de acuerdo con la manera en que **MiClaseGen** está escrita, no es necesario que **T** y **V** sean realmente tipos diferentes. Por ejemplo, es perfectamente correcto (en principio) construir un objeto de **MiClaseGen** como se muestra aquí:

```
MiClaseGen<String, String> objetivo = new MiClaseGen<String, String>()
```

En este caso, **T** y **V** serán reemplazados con **String**, lo que hace que ambas versiones de **set( )** sean idénticas. Esto es, por supuesto, un error.

En segundo lugar, y más importante aún, el borrado de tipo de **set( )** reduce ambas versiones a lo siguiente:

```
void set (Object o) { // ...
```

Por lo tanto, la sobrecarga de **set( )**, tal y como se intenta llevar a cabo en **MiClaseGen,** es inherentemente ambigua. La solución en este caso es usar dos nombres de métodos separados, en lugar de tratar de sobrecargar **set( )**.

Comprobación de avance

1. El borrado _____ todos los parámetros de tipo, sustituyendo sus tipos límite y aplicando moldeos apropiados.
2. Como opción predeterminada, el tipo límite de un parámetro de tipo es _____.
3. La ambigüedad puede presentarse cuando el borrado de tipo hace que dos declaraciones aparentemente diferentes se resuelvan en el mismo tipo borrado. ¿Cierto o falso?

HABILIDAD FUNDAMENTAL 13.12

Algunas restricciones genéricas

Debe tomar en cuenta unas cuantas restricciones al utilizar elementos genéricos. Éstas incluyen la creación de objetos de un parámetro de tipo, miembros estáticos, excepciones y matrices. Cada una se examinará a continuación.

No pueden crearse instancias de parámetros de tipo

No es posible crear una instancia de un parámetro de tipo. Por ejemplo, revise esta clase:

```
// No puede crearse una instancia de T.
class Gen<T> {
  T ob;
  Gen() {
    ob = new T(); // ¡¡¡Ilegal!!!
  }
}
```

En este caso, resulta ilegal tratar de crear una instancia de **T**, lo cual es fácil comprender: como no existe **T** en tiempo de ejecución, ¿de qué forma podría saber el compilador qué tipo de objeto crear? Recuerde que el borrado elimina todos los parámetros de tipo durante el proceso de compilación.

1. removes
2. **Objeto**
3. True

Restricciones en miembros estáticos

Ningún miembro **static** puede usar un parámetro de tipo que esté declarado por la clase que la incluye. Por ejemplo, todos los miembros **static** de esta clase son ilegales:

```
class Incorrecta<T> {
  // Incorrecta, no hay variables static de tipo T.
  static T ob;

  // Incorrecta, ningún método static puede usar T.
  static T obtenerob() {
    return ob;
  }

  // Incorrecta, ningún método static puede accesar el objeto
  // de tipo T.
  static void mostrarob() {
    System.out.println(ob);
 }
}
```

Aunque no puede declarar miembros **static** que usen un parámetro de tipo que esté declarado por la clase que la incluye, *sí puede* declarar métodos genéricos **static**, los cuales definen sus propios parámetros de tipo, como se realizó al principio de este capítulo.

Restricciones genéricas de matriz

Existen dos restricciones importantes respecto a los elementos genéricos que se aplican a las matrices. En primer lugar, no puede crear instancias de una matriz cuyo tipo de base sea un parámetro de tipo. En segundo lugar, no puede crear una matriz de referencias genéricas específicas de tipo. El siguiente programa corto muestra ambas situaciones.

```
// Genéricos y matrices.
class Gen<T extends Number> {
  T ob;

  T vals[]; // OK

  Gen(T o, T[] nums) {
    ob = o;

    // Esta instrucción es ilegal.
//  vals = new T[10]; // No puede crear una matriz de T

    // Pero esta instrucción es correcta.
```

```
    vals = nums; // Correcto asignar una referencia a una matriz existente
  }
}

class MatricesGen {
  public static void main(String args[]) {
    Integer n[] = { 1, 2, 3, 4, 5 };

    Gen<Integer> iOb = new Gen<Integer>(50, n);

    // No puede crear una matriz de referencias genéricas a tipos
específicos.
    // Gen<Integer> gens[] = new Gen<Integer>[10]; // ¡Incorrecta!

    // Esto es correcto.
    Gen<?> gens[] = new Gen<?>[10]; // Correcto
  }
}
```

Como lo muestra este programa, resulta válido declarar una referencia a una matriz de tipo **T**, como en esta línea:

```
T vals[]  // Correcto
```

Sin embargo, no puede crear una instancia de una matriz de **T**, como se intenta en esta línea convertida en comentario:

```
//  vals = new T[10]; // No puede crear una matriz de T
```

La razón por la que no puede crear una matriz de **T** es que **T** no existe en tiempo de ejecución, de modo que no hay manera de que el compilador sepa qué tipo de matriz crear en realidad.

Sin embargo, puede pasar a **Gen()** una referencia a una matriz compatible con el tipo al crear un objeto, y asignar esa referencia a **vals**, como lo hace el programa en la siguiente línea:

```
vals = nums; // Correcto asignar una referencia a una matriz existente
```

Esto funciona porque la matriz pasada a **Gen** tiene un tipo conocido, que será el mismo que **T** en el tiempo de la creación del objeto.

Dentro de **main()**, observe que no puede declarar una matriz de referencias a un tipo genérico específico. Es decir, esta línea

```
// Gen<Integer> gens[] = new Gen<Integer>[10]; // ¡Incorrecta!
```

no se compilará.

Restricción de excepciones genéricas

Una clase genérica no puede extender **Throwable**, lo que significa que no puede crear clases de excepciones genéricas.

Continuación del estudio de los elementos genéricos

Como se mencionó al principio, en este módulo se proporcionan los conocimientos suficientes para utilizar los elementos genéricos de manera efectiva en sus propios programas. Sin embargo, hay muchos temas y casos especiales que no se cubren aquí. Los lectores que tengan interés especial en los elementos genéricos tal vez querrán aprender acerca de la manera en que afectan, por ejemplo, a las jerarquías de clases, a las comparaciones de tipo en tiempo de ejecución y a la sobrescritura. Un análisis de éstos y otros temas se encuentra en el libro *Java: The Complete Reference, j2SE 5 Edition* (McGraw-Hill/Osborne, 2005).

Comprobación de dominio del módulo 13

1. Los elementos genéricos constituyen una adición importante a Java porque permiten la creación de código:
   a) Con seguridad de tipo
   b) Reciclable
   c) Confiable
   d) Todo lo anterior

2. ¿Puede usarse un tipo primitivo como un argumento de tipo?

3. Muestre cómo declarar una clase llamada **CalendarioVuelo** que tome dos parámetros genéricos.

4. Partiendo de su respuesta a la pregunta 3, cambie el segundo parámetro de tipo de **CalendarioVuelo** para que pueda extender **Thread**.

5. Ahora, cambie **CalendarioVuelo** para que su segundo parámetro de tipo sea una subclase de su primer parámetro de tipo.

6. En relación con los elementos genéricos, ¿qué es y qué hace?

7. ¿Es posible limitar el argumento del comodín?

8. Un método genérico llamado **MiGen()** cuenta con un parámetro de tipo. Más aún, **MiGen()** tiene un parámetro cuyo tipo es el del parámetro de tipo. Asimismo, devuelve un objeto de ese parámetro de tipo. Demuestre cómo declarar **MiGen()**.

9. Dada esta interfaz genérica

```
interface IfIGen<T, extends T> { // ...
```

muestre la declaración de una clase llamada **MiClase** que implementa **IFIGen**.

10. Dada una clase genérica llamada **Contador<T>**, muestre cómo crear un objeto de su tipo bruto.

11. ¿Los parámetros de tipo existen en el tiempo de ejecución?

12. Convierta su solución a la pregunta 10 de la Comprobación de dominio del módulo 9 para que sea genérica. En el proceso, cree una interfaz de pila llamada **PilaIGen** que defina genéricamente las operaciones **empujar()** y **jalar()**.

Módulo 14

Applets, eventos y temas diversos

HABILIDADES FUNDAMENTALES

El objetivo principal de este libro es la enseñanza de los elementos del lenguaje Java y, en este sentido, estamos casi por terminar. En los 13 módulos anteriores nos hemos concentrado en las funciones de Java que se encuentran definidas por el lenguaje, como sus palabras clave, su sintaxis, la estructura de bloques, las reglas de conversión de tipos, etc. En este momento, ya tiene el conocimiento suficiente para escribir programas sofisticados y útiles de Java. Sin embargo, hay una parte importante de la programación en Java que requiere más que una simple comprensión del propio lenguaje: *el applet*. El applet (o subprograma) es el tipo más importante de aplicación de Java, así que ningún libro sobre Java estaría completo si no lo cubriera. Por lo tanto, en este módulo se presenta una revisión general de la programación de un applet.

Los applets emplean una arquitectura única y requieren el uso de varias técnicas especiales de programación. Una de estas técnicas es el *manejo de eventos*. Los eventos son la manera en que un applet recibe entradas del mundo exterior. Debido a que el manejo de eventos constituye una parte importante de casi todos los applets, se presentará también a continuación.

Esté prevenido: los applets y el manejo de eventos son temas muy amplios. Una cobertura completa y detallada de éstos queda fuera del alcance de este libro. Aquí aprenderá sus fundamentos y estudiará varios ejemplos, pero sólo veremos los aspectos más generales. Sin embargo, al final de este módulo, contará con una base sólida para empezar el estudio a profundidad de estos importantes temas.

Este módulo termina con la descripción de unas cuantas palabras clave, como **instanceof** y **native**, que no se han descrito en ninguna otra parte del libro. Estas palabras clave se usan para una programación más avanzada, pero se resumen a continuación con el fin de que la información esté completa.

HABILIDAD FUNDAMENTAL 14.1

Fundamentos de los applets

Los applets son diferentes respecto al tipo de programas que se mostró en los módulos anteriores. Como se mencionó en el módulo 1, los applets son pequeños programas que están diseñados para transmitirse por medio de Internet y para ejecutarse dentro de un explorador. Debido a que la máquina virtual de Java se encarga de ejecutar todos los programas de Java, incluidos los applets, éstos ofrecen una manera segura de descargar y ejecutar programas dinámicamente en Web. Antes de analizar cualquier teoría o cualquier detalle relacionado, empecemos por examinar un applet simple. Éste realiza una sola función: despliega la cadena, “Java facilita los applets”, dentro de una ventana.

```
// Un applet mínimo.
import java.awt.*;
import java.applet.*;

public class AppletSimple extends Applet {
  public void paint(Graphics g) {
    g.drawString("Java facilita los applets.", 20, 20);
  }
}
```

Observe estas instrucciones **import**. Todos los applets las usan.

Esto da salida a la ventana del applet.

Este applet empieza con dos instrucciones **import**. La primera importa las clases del juego de herramientas abstractas de ventanas (Abstract Window Toolkit, AWT). Los applets interactúan con el usuario a través del AWT, no de las clases de E/S basadas en la consola. El AWT contiene soporte para una interfaz gráfica basada en ventanas. Como es de esperarse, éste es muy largo y complejo. Un análisis completo del AWT requeriría un libro completo. Por fortuna, como sólo estaremos creando applets muy simples, sólo haremos un uso limitado del AWT. La siguiente instrucción **import** importa el paquete **Applet**. Este paquete contiene la clase **Applet**. Cada applet que llegue a crear debe ser una subclase de **Applet**.

La siguiente línea del programa declara la clase **AppletSimple**. Esta clase debe declararse como **public** porque será accedida por código del exterior.

Dentro de **AppletSimple**, se declara **paint( )**. Este método está definido por la clase **Component** de AWT (que es una superclase de **Applet**) y el applet debe sobrescribirlo. Se llama a **paint( )** cada vez que el applet debe volver a desplegar su salida. Esto puede ocurrir por varias razones. Por ejemplo, otra ventana podría sobrescribir la ventana en que el applet se está ejecutando y luego descubrirla. O bien, puede minimizarse la ventana del applet y luego restablecerla. También se llama a **paint( )** cuando la ejecución del applet inicia. Cualquiera que sea la causa, cada vez que se debe redibujar la salida del applet, se llama a **paint( )**. El método **paint( )** tiene un parámetro de tipo **Graphics**. Dicho parámetro contendrá el contexto gráfico, el cual describe el entorno gráfico en que el applet se está ejecutando. Este contexto se usa cada vez que se requiere salida al applet.

Dentro de **paint( )**, hay una llamada a **drawString( )**, que es un miembro de la clase **Graphics**. Este método da salida a una cadena que empieza en la ubicación especificada X,Y. Ésta tiene la siguiente forma general:

```
void drawString(String mensaje, int x, int y)
```

Aquí, *mensaje* es la cadena que tendrá salida a partir de *x*, *y*. En una ventana de Java, la esquina superior izquierda es la ubicación 0,0. La llamada a **drawString( )** en el applet hace que el mensaje se despliegue a partir de la ubicación 20,20.

Observe que el applet no tiene un método **main( )**. A diferencia de los programas que se han mostrado en este libro, los applets no inician la ejecución cuando el nombre de su clase se pasa a un explorador u otro programa habilitado por applets.

Después de que ha introducido el código fuente de **AppletSimple**, debe compilarlo de la misma manera en la que ha compilado sus programas. Sin embargo, la ejecución de **AppletSimple** incluye un proceso diferente. Hay dos maneras en las que puede ejecutar un applet: dentro de un explorador o mediante una herramienta de desarrollo especial que despliega applets. La herramienta que se proporciona con el JDK de Java estándar es el **appletviewer**, el cual usaremos para ejecutar los applets desarrollados de este módulo. Por supuesto, también puede ejecutarlos en su explorador, pero el **appletviewer** es mucho más fácil de usar durante el desarrollo.

Para ejecutar un applet (en un explorador Web o en el **appletviewer**), necesita escribir un archivo de texto corto de HTML que contenga la etiqueta APPLET apropiada. (También puede usar la etiqueta

más reciente OBJECT; sin embargo, en este libro se usará APPLET pues es el método tradicional.) He aquí el archivo HTML que ejecuta **AppletSimple**.

```
<applet code="AppletSimple" width=200 height=60>
</applet>
```

Las instrucciones **width** y **height** especifican las dimensiones del área de despliegue empleada por el applet.

Para ejecutar **AppletSimple** con un visor de applets, deberá ejecutar este archivo HTML. Por ejemplo, si el archivo HTML anterior se llama **IniciarApp.html**, entonces la siguiente línea de comandos ejecutará **AppletSimple**:

```
C:\>appletviewer IniciarApp.html
```

Aunque el uso de un archivo HTML independiente para ejecutar un applet no resulta incorrecto, existe una manera más fácil. Simplemente incluya un comentario cerca de la parte superior de su archivo de código fuente del applet que contiene la etiqueta APPLET. Si usa este método, el archivo fuente **AppletSimple** tendrá este aspecto:

```
import java.awt.*;
import java.applet.*;
/*
<applet code="AppletSimple" width=200 height=60>
</applet>
*/

public class AppletSimple extends Applet {
  public void paint(Graphics g) {
    g.drawString("Java facilita los applets", 20, 20);
  }
}
```

Appletviewer utiliza este HTML para ejecutar el applet.

Ahora podrá ejecutar el applet al pasar el nombre de su archivo fuente al **appletviewer**. Por ejemplo, esta línea de comandos desplegará ahora **AppletSimple**.

```
C:\appletviewer AppletSimple.java
```

La ventana producida por **AppletSimple**, tal y como el **appletviewer** la despliega, se muestra en la siguiente ilustración:

Cuando use **appletviewer**, tenga en cuenta que éste proporciona el marco de la ventana. Los applets que se ejecutan en el explorador no tendrán un marco visible.

Revisemos los puntos clave de un applet:

- Todos los applets son subclases de **Applet**.
- Los applets no necesitan un método **main( )**.
- Los applets deben ejecutarse bajo un visor de applets o un explorador compatible con Java.
- La E/S de usuario no se realiza con las clases de E/S de flujo de Java. En cambio, los applets usan la interfaz proporcionada por el AWT.

Comprobación de avance

1. ¿Qué es un applet?

2. ¿Qué método da salida a la ventana del applet?

3. ¿Qué paquete debe incluirse cuando se crea un applet?

4. ¿Cómo se ejecutan los applets?

1. Un applet es un tipo especial de programa de Java que está diseñado para transmitirse mediante Internet y que se ejecuta dentro de un explorador.
2. El método **paint()** despliega salida en la ventana de un applet.
3. El paquete **java.applet** debe incluirse cuando se crea un applet.
4. Los applets se ejecutan en un explorador o mediante herramientas especiales, como **appletviewer**.

Pregunte al experto

P: **He escuchado de algo llamado Swing. ¿Qué es y cómo se relaciona con el AWT?**

R: Swing es un conjunto de clases que proporciona alternativas poderosas y flexibles a los componentes estándar del AWT. Por ejemplo, además de los componentes familiares, como botones, casillas de verificación y etiquetas, Swing proporciona varias adiciones estimulantes, como paneles de fichas, paneles desplazables, árboles y tablas. Incluso componentes familiares como los botones tienen más funciones en Swing. Por ello, muchos programadores de Java usan Swing cuando crean applets.

A diferencia de los componentes de AWT, los de Swing no son implementados por código específico de la plataforma. En cambio, están escritos por completo en Java y, por consiguiente, son independientes de la plataforma. El término *peso ligero* se usa para describir estos elementos.

Aunque los componentes de Swing ofrecen alternativas a las proporcionadas por el AWT, es importante comprender que Swing está construido sobre la base del AWT. En consecuencia, se requiere una comprensión sólida del AWT para usar Swing de manera efectiva.

Organización del applet y elementos esenciales

Aunque el applet anterior es completamente válido, un applet así de simple tiene poco valor. Antes de que pueda crear applets útiles, debe saber más acerca de la manera en que los applets están organizados, los métodos que usan y la manera en la que interactúan con el sistema en tiempo de ejecución.

HABILIDAD FUNDAMENTAL 14.2

La arquitectura del applet

Un applet es un programa con base en ventanas. Como tal, su arquitectura es diferente a la de los programas con base en la consola que se mostraron en la primera parte de este libro. Si está familiarizado con la programación en Windows, se sentirá a gusto escribiendo applets. Si no es así, entonces deberá comprender algunos conceptos fundamentales.

En primer lugar, los applets están orientados a eventos. Un applet parece un conjunto de rutinas de servicio de interrupción. He aquí cómo funciona el proceso: un applet espera hasta que un evento ocurra. El sistema en tiempo de ejecución notifica al applet acerca de un evento llamando a un manejador de eventos que haya sido proporcionado por el applet. Una vez que esto sucede, el applet debe tomar las acciones apropiadas y luego regresar rápidamente el control al sistema. Éste es un punto crucial. En su mayor parte, su applet no debe entrar en "modo" de operación, en el cual mantiene el control por un periodo extenso. En cambio, debe realizar acciones específicas como respuesta a eventos y luego regresar el control al sistema de tiempo de ejecución. En las situaciones

en las que su applet necesita realizar una tarea repetitiva por su cuenta (por ejemplo, desplegar un mensaje que recorra su ventana), debe empezar un subproceso adicional de ejecución.

En segundo lugar, es el usuario quien inicia la interacción con un applet (no a la inversa). En un programa basado en la consola, cuando el programa necesita entrada, se la pedirá al usuario y luego llamará a algún método de entrada. De esa manera no funciona en un applet. En cambio, el usuario interactúa con el applet cómo quiere y cuándo quiere. Estas interacciones se envían al applet como eventos a los que éste debe responder. Por ejemplo, cuando el usuario hace clic dentro de la ventana de un applet, se genera un evento de clic del ratón. Si el usuario oprime una tecla mientras la ventana del applet tiene el enfoque de la entrada, se genera un evento de tecla oprimida. Los applet pueden contener varios controles, como botones para oprimir o casillas de verificación. Cuando el usuario interactúa con uno de estos controles, el evento se genera.

Aunque la arquitectura de un applet no es tan fácil de comprender como la de un programa basado en la consola, Java lo simplifica lo más posible. Si ha escrito programas para Windows, seguramente sabe entonces lo intimidante que ese entorno puede resultar. Por fortuna, Java proporciona un método mucho más limpio que se domina con mayor rapidez.

HABILIDAD FUNDAMENTAL 14.3

Esqueleto completo de un applet

Aunque **AppletSimple**, el cual ya se mostró, es un verdadero applet, no contiene todos los elementos requeridos por la mayor parte de los applets. En realidad, todos los applets, excepto los triviales, sobrescriben un conjunto de métodos que proporcionan el mecanismo básico mediante el cual un explorador o un visor de applets hacen interfaz con el applet y controla su ejecución. Cuatro de estos métodos, **init()**, **start()**, **stop()** y **destroy()**, están definidos por **Applet**. El quinto método, **paint()**, que ya ha visto, se hereda de la clase **Component** del AWT. Debido a que se proporciona la implementación predeterminada para todos estos métodos, los applets no necesitan sobrescribir los métodos que no emplean. Estos cinco métodos pueden ensamblarse en el esqueleto mostrado aquí:

```
// Esqueleto de un applet.
import java.awt.*;
import java.applet.*;
/*
<applet code="EsqueletoApplet" width=300 height=100>
</applet>
*/

public class EsqueletoApplet extends Applet {
  // Llamado primero.
  public void init() {
    // inicialización
  }

  /* Llamado segundo, después de init().  También llamado ahora
```

```
    se le llama cada que se reinicia el applet. */
  public void start() {
    // inicia o reanuda la ejecución
  }

  // Llamado cuando se detiene el applet.
  public void stop() {
    // suspende la ejecución
  }

  /* Llamado cuando se termina el applet. Es el
     Último método ejecutado. */
  public void destroy() {
    // realiza actividadesde cierre
  }

  // Llamado cuando una ventana de un applet debe restablecerse.
  public void paint(Graphics g) {
    // vuelva a desplegar el contenido de la ventana
  }
}
```

Aunque este esqueleto no lleva a cabo nada, puede compilarse y ejecutarse. De ahí que pueda utilizarlo como punto de partida para los applets que cree.

HABILIDAD FUNDAMENTAL 14.4

Inicialización y terminación de applets

Es importante comprender el orden en que se ejecutan los diversos métodos mostrados en el esqueleto. Cuando empieza un applet, se llama a los siguientes métodos en esta secuencia:

1. **init( )**
2. **start( )**
3. **paint( )**

Cuando el applet se termina, se presenta la siguiente secuencia del método:

1. **stop( )**
2. **destroy( )**

Revisemos más de cerca estos métodos.

El método **init( )** es el primero en ser llamado. En **init( )**, su applet inicializará variables y realizará otras actividades de inicio.

El método **start( )** es llamado después de **init( )**. También se le llama para reiniciar un applet después de que éste se ha detenido, como cuando el usuario regresa a una página Web previamente desplegada que contenga un applet. Por lo tanto, **start( )** podría ser llamado más de una vez durante el ciclo de vida de un applet.

El método **paint( )** es llamado cada vez que la salida de su applet debe ser redibujada y fue descrita antes.

Cuando se abandona la página que contiene su applet, se llama al método **stop( )**. Usará **stop( )** para suspender cualquier subproceso descendiente creado por el applet y para realizar cualquier otra actividad que sea requerida para colocar el applet en un estado seguro e inactivo. Recuerde que una llamada a **stop( )** no significa que el applet debe terminarse, ya que podría reiniciarse con una llamada a **start( )** si el usuario regresa a la página.

El método **destroy( )** es llamado cuando ya no se requiere el applet. Se usa para realizar cualquier cierre de operaciones requeridas por el applet.

Comprobación de avance

1. ¿Cuáles son los cinco métodos que sobrescribirán la mayor parte de los applets?
2. ¿Qué debe hacer su applet cuando se llama a **start( )**?
3. ¿Qué debe hacer su applet cuando se llama a **stop( )**?

HABILIDAD FUNDAMENTAL 14.5

Solicitud de repintado

Como regla general, un applet sólo escribe en su ventana cuando el sistema en tiempo de ejecución llama al método **paint( )**. Esto plantea una pregunta interesante: ¿cómo puede causar el propio applet que se actualice su ventana cuando su información cambia? Por ejemplo, si un applet está desplegando un letrero en movimiento, ¿qué mecanismo debe usar el applet para actualizar la ventana cada vez que este letrero se desplaza? Recuerde que una de las restricciones arquitectónicas fundamentales que se le imponen a un applet es que debe regresar rápidamente el control al sistema en tiempo de ejecución de Java. Por ejemplo, no puede crear un bucle dentro de **paint( )** que desplace repetidamente el letrero. Esto evitaría que el control regrese al sistema en tiempo de ejecución. Dada esta restricción, parecería que, en el mejor de los casos, la salida a la ventana de su applet resulta difícil. Por fortuna, esto no es así. Cada vez que su applet necesite actualizar la información desplegada en su ventana, simplemente llama a **repaint( )**.

1. Los cinco métodos son **init( )**, **start( )**, **stop( )**, **destroy( )** y **paint( )**.
2. Cuando se llama a **start( )**, el applet debe iniciarse o reiniciarse.
3. Cuando **stop( )** es llamado, el applet debe ponerse en pausa.

El método **repaint( )** está definido por la clase **Component** de AWT. Éste hace que el sistema en tiempo de ejecución ejecute una llamada al método **update( )** de su applet que, en su implementación predeterminada, llama a **paint( )**. Por tanto, para que otra parte de su applet dé salida a su ventana, simplemente almacene la salida y luego llame a **repaint( )**. Esto genera una llamada a **paint( )**, que puede desplegar la información almacenada. Por ejemplo, si parte de su applet necesita dar salida a una cadena, puede almacenar esta cadena en una variable **String** y luego llamar a **repaint( )**. Dentro de **paint( )**, dará salida a la cadena usando **drawString( )**.

A continuación se muestra la versión más simple de **repaint( )**:

```
void repaint( )
```

Esta versión hace que toda la ventana se vuelva a pintar.

Otra versión de **repaint( )** especifica una región que se volverá a pintar:

```
void repaint(int izquierda, int arriba, int ancho, int alto)
```

Aquí, las coordenadas de la esquina superior izquierda de la región están especificadas por *izquierda* y *arriba*, y el ancho y la altura de la región se pasan en *ancho* y *alto*. Estas dimensiones se especifican en píxeles. Usted ahorra tiempo al especificar que se repinte una región puesto que las actualizaciones de la ventana son costosas en relación con el tiempo. Si sólo necesita actualizar una pequeña parte de la ventana, le resultará más eficiente repintar sólo esa región.

El proyecto 14.1 contiene un ejemplo que demuestra a **repaint( )**.

El método update( )

Hay otro método que se relaciona con el repintado. Se le conoce como **update( )** y tal vez su applet quiera sobrescribirlo. Este método está definido por la clase **Component** y es llamado cuando su applet solicita que una parte de su ventana se redibuje. La versión predeterminada de **update( )** simplemente llama a **paint( )**. Sin embargo, puede sobrescribir el método **update( )** para que realice un repintado más sutil. Sin embargo, esta técnica avanzada está más allá del alcance de este libro.

Pregunte al experto

P: **Es posible que un método diferente de** paint( ) **o** update( ) **dé salida a la ventana de un applet?**

R: Sí. Para ello debe obtener un contexto gráfico llamando a **getGraphics( )** (el cual está definido por **Component**) y luego usar este contexto para salir a la ventana. Sin embargo, para casi todas las aplicaciones, resulta mejor y más fácil enrutar la salida a la ventana a través de **paint( )** y llamar a **repaint( )** cuando el contenido de la ventana cambie.

Proyecto 14.1 Un applet simple de letrero

Letrero.java

Para demostrar **repaint()**, se presenta un applet simple de letrero. Este applet desplaza un mensaje, de derecha a izquierda, a través de la ventana del applet. Debido a que el desplazamiento del mensaje resulta una tarea repetitiva, se lleva a cabo mediante un subproceso separado, creado por el applet cuando éste se inicializa. Los letreros son funciones populares de Web. Este proyecto muestra la manera en que se usa un applet de Java para crear uno.

Paso a paso

1. Cree un archivo llamado **Letrero.java**.
2. Empiece por crear el applet de letrero con las siguientes líneas.

```
/*
   Proyecto 14.1

   Un applet simple de letrero.

   Este applet crea un subproceso que desplaza
   el mensaje contenido en msj de derecha a izquierda
   a través de la ventana del applet.
*/
import java.awt.*;
import java.applet.*;
/*
<applet code="Letrero" width=300 height=50>
</applet>
*/

public class Letrero extends Applet implements Runnable {
  String msj = " Java gobierna Web ";
  Thread t;
  boolean marcaAlto;

  // Inicializa t en null.
  public void init() {
    t = null;
  }
```

Observe que, como se esperaba, **Letrero** extiende **Applet** pero también implementa **Runnable**, lo cual es necesario debido a que el applet creará un segundo subproceso de ejecución que se usará para desplazar el letrero. El mensaje que se desplazará en el letrero está contenido en la variable **msj** de **String**. Una referencia al subproceso que ejecuta el applet se almacena en **t**. La variable **boolean marcaAlto** se usa para detener el applet. Dentro de **init()**, la variable de referencia al subproceso **t** tiene el valor **null**.

(continúa)

3. Agregue el método **start( )** que se muestra a continuación.

```
// Inicia subproceso
public void start() {
  t = new Thread(this);
  marcaAlto = false;
  t.start();
}
```

El sistema en tiempo de ejecución llama a **start( )** para iniciar la ejecución del applet. Dentro de **start( )**, se crea un nuevo subproceso de ejecución que se asigna a la variable **t** de **Thread**. Luego, se asigna **false** a **marcaAlto**. A continuación, una llamada a **t.start( )** inicializa el subproceso. Recuerde que **t.start( )** llama a un método definido por **Thread**, que hace que **run( )** inicie su ejecución. De manera que no genera una llamada a la versión de **start( )** definida por **Applet**, pues son dos métodos separados.

4. Agregue el método **run( )**, como se muestra aquí.

```
// Punto entrada del subproceso que ejecuta el letrero.
public void run() {
  char car;

  // Despliega letrero
  for( ; ; ) {
    try {
      repaint();
      Thread.sleep(250);
      car = msj.charAt(0);
      msj = msj.substring(1, msj.length());
      msj += car;
      if(marcaAlto)
        break;
    } catch(InterruptedException exc) {}
  }
}
```

En **run( )** los caracteres de la cadena contenida en **msj** se rotan de manera repetida a la izquierda. Entre cada rotación, se hace una llamada a **repaint( )**. Esto causa que en algún momento se llame al método **paint( )** y se despliegue el contenido actual de **msj**. Entre cada iteración, **run( )** duerme por un cuarto de segundo. El efecto neto de **run( )** es que el contenido de **msj** se desplaza de derecha a izquierda en un despliegue en movimiento constante. En cada iteración, se revisa la variable **marcaAlto**. Cuando es **true**, el método **run( )** termina.

5. Agregue el código para **stop( )** y **paint( )**, como se muestra aquí.

```
// Pone en pausa el letrero.
public void stop() {
  marcaAlto = true;
```

```
  t = null;
}

// Despliega el letrero.
public void paint(Graphics g) {
  g.drawString(msj, 50, 30);
}
```

Si un explorador está desplegando el applet cuando se ve una nueva página, se llama al método **stop()**, que establece **marcaAlto** en **true**, lo que causa que **run()** termine. De igual forma establece **t** en **null**. Por consiguiente, ya no hay una referencia al objeto de **Thread**, y éste puede reciclarse la siguiente ocasión en que se ejecute el recolector de basura. Este es el mecanismo usado para detener el subproceso cuando su página ya no se encuentra a la vista. Cuando el applet se vuelve a traer a la vista, se llama una vez más a **start()**, lo que inicia un nuevo subproceso para ejecutar el letrero.

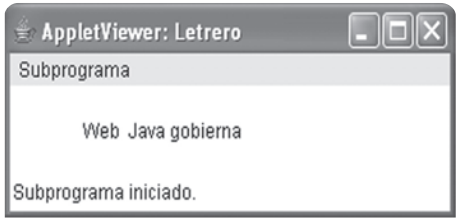
6. Todo el applet de letrero se muestra aquí.

```
/*
   Proyecto 14.1

   Un applet simple de letrero.

   Este applet crea un subproceso que desplaza
   el mensaje contenido en msj de derecha a izquierda
   a través de la ventana del applet.
*/
import java.awt.*;
import java.applet.*;
/*
<applet code="Letrero" width=300 height=50>
</applet>
*/

public class Letrero extends Applet implements Runnable {
  String msj = " Java gobierna Web ";
  Thread t;
  boolean marcaAlto;

  // Inicializa t en null.
  public void init() {
    t = null;
  }

  // Inicia subproceso
  public void start() {
    t = new Thread(this);
```

(continúa)

```
    marcaAlto = false;
    t.start();
  }

  // Punto de entrada del subproceso que ejecuta el letrero.
  public void run() {
    char car;

    // Despliega letrero
    for( ; ; ) {
      try {
        repaint();
        Thread.sleep(250);
        car = msj.charAt(0);
        msj = msj.substring(1, msj.length());
        msj += car;
        if(marcaAlto)
          break;
      } catch(InterruptedException exc) {}
    }
  }

  // Pone en pausa el letrero.
  public void stop() {
    marcaAlto = true;
    t = null;
  }

  // Despliega el letrero.
  public void paint(Graphics g) {
    g.drawString(msj, 50, 30);
  }
}
```

Aquí se muestra una salida de ejemplo:



HABILIDAD FUNDAMENTAL 14.6

Uso de la ventana de estado

Además de desplegar información en su ventana, un applet puede dar también salida a un mensaje en la ventana de estado del explorador o el visor de applets en que se esté ejecutando. Para ello, se llama a **showStatus()**, que está definido por **Applet**, con la cadena que quiera desplegar. La forma general de **showStatus()** se muestra a continuación:

void showStatus(String *msj*)

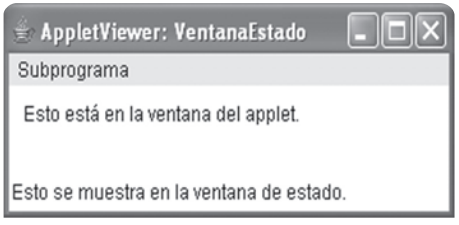
Aquí, *msj* es la cadena que se habrá de desplegar.

La ventana de estado es un buen lugar para retroalimentar al usuario acerca de lo que está ocurriendo en el applet, sugerir opciones o reportar algunos tipos de errores. La ventana de estado también representa una excelente ayuda para la depuración porque le ofrece una manera fácil de dar salida a la información acerca de su applet.

El siguiente applet demuestra **showStatus()**:

```
// Uso de la ventana de estado.
import java.awt.*;
import java.applet.*;
/*
<applet code="VentanaEstado" width=300 height=50>
</applet>
*/

public class VentanaEstado extends Applet{
  // Despliega msj en la ventana del applet.
  public void paint(Graphics g) {
    g.drawString("Esto está en la ventana del applet.", 10, 20);
    showStatus("Esto se muestra en la ventana de estado.");
  }
}
```

Aquí se muestra la salida de ejemplo de este programa:



HABILIDAD FUNDAMENTAL 14.7

Paso de parámetros a applets

Puede pasar parámetros a su applet. Para ello, debe usar el atributo PARAM de la etiqueta APPLET especificando el nombre y valor del parámetro. Para recuperar un parámetro, debe usar el método **getParameter( )**, definido por **Applet**. En seguida se muestra su forma general:

String getParameter(String *nombreParam*)

Aquí, *nombreParam* es el nombre del parámetro y regresa el valor del parámetro especificado en la forma de un objeto de **String**. Por lo tanto, para valores numéricos y **boolean**, necesitará convertir sus representaciones de cadena en sus formatos internos. Si no puede encontrarse el parámetro especificado, se regresa **null**. Asegúrese entonces de confirmar que el valor regresado por **getParameter( )** sea válido. Además, compruebe cualquier parámetro que se convierta en un valor numérico, confirmando que una conversión válida tuvo lugar.

He aquí un ejemplo que demuestra el paso de parámetros:

```
// Paso de un parámetro a un applet.
import java.awt.*;
import java.applet.*;

/*
<applet code="Param" width=300 height=80>
<param name=autor value="Herb Schildt">
<param name=objetivo value="Demuestra Parámetros">
<param name=versión value=2>
</applet>
*/

public class Param extends Applet {
  String autor;
  String objetivo;
  int ver;

  public void start() {
    String temp;

    autor = getParameter("autor");
    if(autor == null) autor = "no se encontró";

    objetivo = getParameter("objetivo");
    if(objetivo == null) objetivo = "no se encontró";

    temp = getParameter("versión");
```

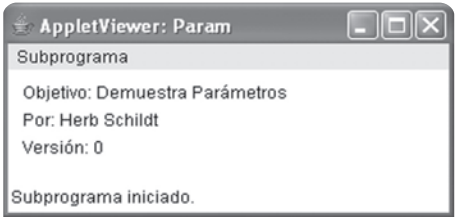
Estos parámetros de **HTML** se pasan al applet.

¡Es importante confirmar que existe el parámetro!

```
    try {
      if(temp != null)
        ver = Integer.parseInt(temp);
      else
        ver = 0;
    } catch(NumberFormatException exc) {
        ver = -1; // código de error
    }
  }

  public void paint(Graphics g) {
    g.drawString("Objetivo: " + objetivo, 10, 20);
    g.drawString("Por: " + autor, 10, 40);
    g.drawString("Versión: " + ver, 10, 60);
  }
}
```

¡También es importante asegurarse de que las conversiones numéricas tienen éxito!

Enseguida se muestra la salida de ejemplo de este programa:



Comprobación de avance

1. ¿Cómo se logra que el método **paint()** del applet sea llamado?
2. ¿En qué lugar **showStatus()** despliega una cadena?
3. ¿Qué método se utiliza para obtener un parámetro especificado en la etiqueta APPLET?

1. Para que se llame a **paint()**, se debe llamar a **repaint()**.
2. **showStatus()** despliega salida en una ventana de estado de un applet.
3. Para obtener un parámetro, debe llamar a **getParameter()**.

HABILIDAD FUNDAMENTAL 14.8

La clase Applet

Como se mencionó, todos los applets son subclases de la clase **Applet**. **Applet** hereda las siguientes superclases definidas por el AWT: **Component**, **Container** y **Panel**. Un applet tiene entonces acceso a toda la funcionalidad del AWT.

Además de los métodos descritos en las secciones anteriores, **Applet** contiene otros que le brindan un control detallado sobre la ejecución de su applet. Todos los métodos definidos por **Applet** se muestran en la tabla 14.1.

Método	Descripción
void destroy()	Llamado por el explorador justo antes de que un applet termine. Su applet sobrescribirá este método si necesita realizar alguna limpieza antes de su destrucción.
AccesibleContext getAccesibleContext()	Regresa el contexto de accesibilidad para el objeto que invoca.
AppletContext getAppletContext()	Regresa el contexto asociado con el applet.
String getAppletInfo()	Regresa una cadena que describe el applet.
AudioClip getAudioClip(URL *url*)	Regresa un objeto de **AudioClip** que encapsula el clip de audio encontrado en la ubicación especificada por *url*.
AudioClip getAudioClip(URL *url*, String *nombreClip*)	Regresa un objeto de **AudioClip** que encapsula el clip de audio encontrado en la ubicación especificada por *url* y que tiene el nombre especificado por *nombreClip*.
URL getCodeBase()	Regresa el URL asociado con el applet que invoca.
URL getDocumentBase()	Regresa el URL del documento HTML que invoca al applet.
Image getImage(URL *url*)	Regresa un objeto de **Image** que encapsula la imagen encontrada en la ubicación especificada por *url*.
Image getImage(URL *url*, String *nombreImagen*)	Regresa un objeto de **Image** que encapsula la imagen encontrada en la ubicación especificada por *url* y que tiene el nombre especificado por *nombreImagen*.
Locale getLocale()	Regresa un objeto de **Locale** que es usado por varias clases y métodos sensibles al lugar en que se presentan los eventos.

Tabla 14.1 Los métodos definidos por **Applet**.

Método	Descripción
String getParameter(String *nombreParam*)	Regresa el parámetro asociado con *nombreParam*. Se regresa **null** si no se encuentra el parámetro especificado.
String[][] getParameterInfo()	Regresa una tabla de **String** que describe los parámetros reconocidos por el applet. Cada entrada de la tabla debe contener tres cadenas que contienen el nombre del parámetro, una descripción de su tipo o rango (o ambos) y una explicación de su objetivo.
void init()	Se llama a este método cuando la ejecución de un applet empieza. Es el primer método llamado por un applet.
boolean isActive()	Regresa **true** si el applet se ha iniciado. Regresa **false** si se ha detenido.
static final AudioClip	Regresa un objeto de **AudioClip** que encapsula el clip de audio newAudioClip(URL url) encontrado en el lugar especificado por *url*. Este método es similar a **getAudioClip()**, con la excepción de que es estático y puede ejecutarse sin la necesidad de un objeto de **Applet**.
void play(URL *url*)	Si se encuentra un clip de audio en el lugar especificado por el *url*, el clip se reproduce.
void play(URL *url* String *nombreClip*)	Si se encuentra un clip de audio en el lugar especificado por el *url* y con el nombre especificado por *nombreClip*, el clip se reproduce.
void resize(Dimension *dim*)	Cambia el tamaño del applet de acuerdo con las dimensiones especificadas por *dim*. **Dimensión** es una clase almacenada dentro de **java.awt**. Contiene dos campos enteros: **width** y **height**.
void resize(int *ancho*, int *altura*)	Cambia el tamaño del applet de acuerdo con la dimensión especificada por *ancho* y *altura*.
final void setStub(AppletStub *objTalón*)	Hace que *objTalón* sea el talón del applet. El sistema en tiempo de ejecución utiliza este método y por lo general no es llamado por su applet. Un *talón* es una pequeña pieza de código que proporciona los vínculos suficientes entre su applet y el explorador.
void showStatus(String *cad*)	Despliega *cad* en la ventana de estado del explorador o el visor de applets. Si el explorador no tiene soporte a ventana de estado, la acción no se realiza.
void start()	Llamado por el explorador cuando un applet debe iniciar (o reanudar) la ejecución. Se le llama automáticamente después de **init()**, cuando un applet inicia por primera vez.
void stop()	Llamado por el explorador para suspender la ejecución de un applet. Una vez detenido, el applet se reinicia cuando el explorador llama a **start()**.

Tabla 14.1 Los métodos definidos por **Applet**. *(continúa)*

Manejo de eventos

Los applets son programas orientados a eventos. El manejo de eventos es el alma de una programación exitosa de applets. La mayor parte de los eventos a los que su applet responderá son generados por el usuario. Estos eventos se pasan a su applet de diversas maneras. El método específico depende del evento real. Hay varios tipos de eventos: los eventos de manejo más común son los generados por el ratón, el teclado y varios controles, como un botón. El paquete **java.awt.event** da soporte a los eventos.

Antes de empezar nuestro análisis del manejo de eventos, se debe tratar el siguiente tema importante: la manera en que un applet maneja los eventos cambió radicalmente entre la versión original de Java (1.0) y las versiones modernas, a partir de la 1.1. Todavía se da soporte al método de manejo de eventos de la versión 1.0, pero no es recomendable para los nuevos programas. Además, se han desautorizado muchos de los métodos que dan soporte al modelo de eventos de la versión 1.0. El método moderno es la manera en que deben manejarse los eventos en todos los nuevos programas. De hecho es el método que se describirá a continuación.

Una vez más, es importante destacar que no es posible analizar por completo el mecanismo de manejo de eventos. Éste es un tema extenso con muchas funciones y muchos atributos especiales, así que un análisis completo estaría más allá del alcance de este libro. Sin embargo, el panorama general presentado aquí le ayudará a empezar a comprenderlo.

HABILIDAD FUNDAMENTAL 14.9

El modelo de evento de delegación

El método moderno para el manejo de eventos se basa en el *modelo de evento de delegación*. Este modelo define los mecanismos consistentes y estándar para generar y procesar eventos. Su concepto es muy simple: un *origen* genera un evento y lo envía a uno o más *escuchas*. En este esquema, el escucha simplemente espera hasta que recibe un evento. Una vez recibido, procesa el evento y luego regresa. La ventaja de este diseño es que la lógica 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 pieza separada de código. En el modelo de evento de delegación, los escuchas deben registrarse con un origen para recibir una notificación de evento.

Eventos

En el modelo de delegación, un evento es un objeto que describe un cambio de estado en un origen. Puede generarse como consecuencia de la interacción de una persona con el elemento en una interfaz gráfica de usuario, como presionar un botón, ingresar un carácter en el teclado, seleccionar un elemento en una lista y hacer clic con el ratón.

Orígenes de eventos

Un origen de evento es un objeto que genera un evento. Un origen debe registrar escuchas para que éstos reciban notificaciones acerca de un tipo específico de evento. Cada tipo de evento tiene su propio método de registro. He aquí la forma general:

public void add*Tipo*Listener(*Tipo*Listener *ee*)

Aquí, *Tipo* es el nombre del evento y *ee* es una referencia al escucha de eventos. Por ejemplo, el método que registra un escucha de eventos del teclado es **addKeyListener( )**. El método que registra un escucha de movimientos del ratón es **addMouseMotionListener( )**. Cuando ocurre un evento, se notifica a todos los escuchas registrados para que todos ellos reciban una copia del objeto de evento.

Un origen debe proporcionar también un método que permita que un escucha deje de registrar un interés en un tipo de evento específico. La forma general de este tipo de método es el siguiente:

public void remove*Tipo*Listener(*Tipo*Listener *ee*)

Aquí, *Tipo* es el nombre del evento y *ee* es una referencia al escucha del evento. Por ejemplo, para eliminar un escucha del teclado, tendría que llamar a **removeKeyListener( )**.

Los métodos que agregan o eliminan escuchas son proporcionados por el origen que genera eventos. Por ejemplo, la clase **Component** proporciona métodos para agregar y eliminar escuchas de eventos de teclado y ratón.

Escuchas de eventos

Un *escucha* es un objeto que recibe una notificación cuando un evento ocurre. Tiene dos requisitos importantes: en primer lugar, debe haberse registrado con uno o más orígenes para recibir una notificación acerca de tipos de eventos específicos; en segundo lugar, debe implementar métodos para recibir y procesar esas notificaciones.

Los métodos que reciben y procesan eventos están definidos en un conjunto de interfaces que se encuentran en **java.awt.event**. Por ejemplo, la interfaz **MouseMotionListener** define métodos que reciben una notificación cuando se arrastra o mueve el ratón. Cualquier objeto puede recibir y procesar uno o ambos eventos si proporciona una implementación de la interfaz.

Clases de eventos

Las clases que representan eventos constituyen la esencia del mecanismo de manejo de eventos de Java. En la raíz de la jerarquía de clases de eventos de Java se encuentra **EventObject**, el cual está contenido en **java.útil**, y es la superclase de todos los eventos. La clase **AWTEvent**, definida

dentro del paquete **java.awt**, es una subclase de **EventObject** y es la superclase (ya sea directa o indirectamente) de todos los eventos con base en AWT usados por el modelo de evento de delegación.

El paquete **java.awt.event** define varios tipos de eventos generados por varios elementos de la interfaz de usuario. En la tabla 14.2 se enumeran los de uso más común y se proporciona una breve descripción del momento en que se generan.

Interfaces de escuchas de eventos

Los escuchas de eventos reciben notificaciones. Los escuchas se crean al implementar una o más de las interfaces definidas por el paquete **java.awt.event**. Cuando un evento ocurre, el origen del evento invoca al método apropiado definido por el escucha y proporciona un objeto de evento como argumento. En la tabla 14.3 se presenta una lista de las interfaces de escuchas de uso común y se proporciona una breve descripción de los métodos que definen.

Clase de evento	Descripción
ActionEvent	Se genera cuando se oprime un botón, se hace doble clic en un elemento de una lista o se selecciona un elemento de un menú.
AdjustmentEvent	Se genera cuando se manipula una barra de desplazamiento.
ComponentEvent	Se genera cuando un componente está oculto, se mueve, cambia de tamaño o se vuelve visible.
ContainerEvent	Se genera cuando se agrega o elimina un componente de un contenedor.
FocusEvent	Se genera cuando un componente obtiene o pierde el enfoque del teclado.
InputEvent	Superclase abstracta para todas las clases de eventos de entrada de componentes.
ItemEvent	Se genera cuando se hace clic en una casilla de verificación o un elemento de una lista; también ocurre cuando se elige una opción o se selecciona o deja de seleccionar un elemento de menú en el que se pueda hacer clic.
KeyEvent	Se genera cuando se recibe una entrada del teclado.
MouseEvent	Se genera cuando se arrastra o mueve el ratón, se hace clic, o se oprime o libera el botón del ratón; también se genera cuando el ratón ingresa o sale de un componente.
TextEvent	Se genera cuando el valor de un área de texto o un campo de texto cambia.
WindowEvent	Se genera cuando se activa, cierra, desactiva, se abre o cierra una ventana, y cuando se le añaden o quitan iconos a ésta.

Tabla 14.2 Las principales clases de eventos en **java.awt.event**.

Interfaz	Descripción
ActionListener	Define un método para recibir eventos de acción. Los eventos de acción son generados por acciones como oprimir botones y menús.
AdjustmentListener	Define un método para recibir eventos de ajuste, como los producidos por una barra de desplazamiento.
ComponentListener	Define cuatro métodos para reconocer cuándo se oculta, mueve, cambia de tamaño o muestra un componente.
ContainerListener	Define dos métodos para reconocer cuándo se agrega o elimina un componente de un contenedor.
FocusListener	Define dos métodos para reconocer cuándo un componente gana o pierde el enfoque del teclado.
ItemListener	Define un método para reconocer cuándo el estado de un elemento cambia. Por ejemplo, una casilla de verificación genera un evento de elemento.
KeyListener	Define tres métodos para reconocer cuándo se oprime una tecla, se suelta o se está escribiendo.
MouseListener	Define cinco métodos para reconocer cuándo se hace clic, cuando el ratón entra o sale de un componente, o cuando se oprime o se libera.
MouseMotionListener	Define dos métodos para reconocer cuándo se arrastra o mueve el ratón.
TextListener	Define un método para reconocer cuándo cambia el valor de un texto.
WindowListener	Define siete métodos para reconocer cuándo se activa, cierra, desactiva, se abre o cierra una ventana, y cuando se le añaden o quitan iconos a ésta.

Tabla 14.3 Interfaces de escuchas de eventos comunes.

Comprobación de avance

1. Explique brevemente la importancia de **EventObject** y **AWTEvent**.
2. ¿Qué es un origen de evento? ¿Qué es un escucha de evento?
3. Los escuchas deben registrarse con un origen para recibir notificaciones de eventos. ¿Cierto o falso?

1. **EventObject( )** es una superclase de todos los eventos. **AWTEvent** es una superclase de todos los eventos de AWT que son manejados por el modelo de evento de delegación.
2. Un origen de evento genera eventos. Un escucha de eventos recibe notificaciones de eventos.
3. Cierto.

HABILIDAD FUNDAMENTAL 14.10

Uso del modelo de evento de delegación

Ahora que ya tiene un panorama general del modelo de evento de delegación y sus diversos componentes, es momento de verlo en la práctica. La programación de applets que usa el modelo de evento de delegación es realmente muy fácil. Sólo siga estos dos pasos:

1. Implemente la interfaz apropiada en el escucha para que reciba el tipo de evento deseado.
2. Implemente código para registrar y dejar de registrar (si es necesario) al escucha como receptor de las notificaciones de eventos.

Recuerde que un origen puede generar varios tipos de eventos. Cada evento debe registrarse por separado. Además, un objeto puede registrarse para recibir varios tipos de eventos, pero éste debe implementar todas las interfaces que se requieran para recibir esos eventos.

Para ver cómo funciona el modelo de delegación en la práctica, revisaremos un ejemplo que maneja uno de los generadores de eventos más comunes: el ratón.

Manejo de eventos del ratón

Para manejar eventos del ratón, debe implementar las interfaces **MouseListener** y **MouseMotionListener**. La interfaz **MouseListener** define cinco métodos. Si se hace clic en un botón del ratón, se invoca a **mouseClicked( )**. Cuando el ratón ingresa en un componente, se llama al método **mouseEntered**; cuando lo abandona, se llama a **mouseExited( )**. Los métodos **mousePressed( )** y **mouseReleased( )** se invocan cuando un botón del ratón se oprime o libera, respectivamente. Las formas generales de estos métodos se muestran a continuación:

void mouseClicked(MouseEvent *er*)

void mouseEntered(MouseEvent *er*)

void mouseExited(MouseEvent *er*)

void mousePressed(MouseEvent *er*)

void mouseReleased(MouseEvent *er*)

La interfaz **MouseMotionListener** define dos métodos. El método **mouseDragged( )** es llamado varias veces cuando se arrastra el ratón. El método **mouseMoved( )** es llamado varias veces mientras se mueve el ratón. Sus formas generales son:

void mouseDragged(MouseEvent *er*)

void mouseMoved(MouseEvent *er*)

El objeto **MouseEvent** pasado en *er* describe el evento. **MouseEvent** define varios métodos que puede usar para obtener información acerca de lo que sucede. Tal vez los métodos de uso más común de **MouseEvent** sean **getX()** y **getY()**. Éstos regresan las coordenadas X y Y del ratón cuando el evento haya ocurrido. Aquí se muestran sus formas:

int getX( )

int getY( )

En el siguiente ejemplo se usarán estos métodos para desplegar la ubicación actual del ratón.

Un applet simple de evento de ratón

El siguiente applet demuestra el manejo de eventos del ratón. El applet despliega las coordenadas actuales del ratón en la ventana de estado del applet. Cada vez que se oprime un botón, la palabra "Abajo" se despliega en el lugar del puntero del ratón. Cada vez que se suelta el ratón, se muestra la palabra "Arriba". Si se hace clic en un botón, el mensaje "Clic del ratón" se despliega en la esquina superior izquierda del área de despliegue del applet.

Cuando el ratón entra o sale de la ventana del applet, se despliega un mensaje en la esquina superior izquierda del área de despliegue del applet. Cuando se arrastra el ratón, se muestra un *, que sigue al puntero del ratón a medida que se arrastra. Observe que las dos variables **mouseX** y **mouseY** almacenan la ubicación del ratón cuando ocurre un evento de ratón oprimido, liberado o arrastrado. Luego, **paint()** emplea estas coordenadas para desplegar la salida en el punto en que esto sucede.

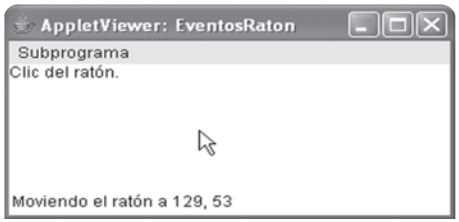
```
// Demuestra los manejadores de eventos de ratón.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="EventosRatón" width=300 height=100>
  </applet>
*/

public class EventosRatón extends Applet
  implements MouseListener, MouseMotionListener {

  String msj = "";
  int mouseX = 0, mouseY = 0; // coordenadas del ratón

  public void init() {
     addMouseListener(this);
     addMouseMotionListener(this); <-- Registra esta clase como escucha de los eventos del ratón.
  }
```

```
// Maneja el clic del ratón.
public void mouseClicked(MouseEvent er) {
  mouseX = 0;
  mouseY = 10;
  msj = "Clic del ratón.";
  repaint();
}

// Maneja entrada del ratón.
public void mouseEntered(MouseEvent er) {
  mouseX = 0;
  mouseY = 10;
  msj = "Entrada del ratón.";
  repaint();
}

// Maneja la salida del ratón.
public void mouseExited(MouseEvent er) {
  mouseX = 0;
  mouseY = 10;
  msj = "Salida del ratón.";
  repaint();
}

// Maneja botón opimido.
public void mousePressed(MouseEvent er) {
  // Guarda las coordenadas
  mouseX = er.getX();
  mouseY = er.getY();
  msj = "Abajo";
  repaint();
}

// Maneja botón liberado.
public void mouseReleased(MouseEvent er) {
  // Guarda coordenadas
  mouseX = er.getX();
  mouseY = er.getY();
  msj = "Arriba";
  repaint();
}

// Maneja ratón arrastrado.
public void mouseDragged(MouseEvent er) {
  // Guarda coordenadas
  mouseX = er.getX();
  mouseY = er.getY();
  msj = "*";
```

Éste y los demás manejadores de eventos responden a los eventos del ratón.

```
    showStatus("Arrastrando el ratón a " + mouseX + ", " + mouseY);
    repaint();
  }

  // Maneja ratón movido.
  public void mouseMoved(MouseEvent er) {
    // muestra estado
    showStatus("Moviendo el ratón a " + er.getX() + ", " +
    er.getY());
  }

  // Despliega msj en la ventana del applet en el lugar X,Y actual.
  public void paint(Graphics g) {
    g.drawString(msj, mouseX, mouseY);
  }
}
```

Aquí se muestra una salida de ejemplo de este programa:



Observemos de cerca este ejemplo: la clase **MouseEvents** extiende **Applet** e implementa las interfaces **MouseListener** y **MouseMotionListener**. Estas dos interfaces contienen métodos que reciben y procesan los diversos tipos de eventos del ratón. Observe que el applet es el origen y el escucha de estos eventos. Esto funciona porque **Component**, que proporciona los métodos **addMouseListener()** y **addMouseMotionListener()**, es una superclase de **Applet**. La función de origen y escucha de eventos es una situación común entre los applets.

Dentro de **init()**, el applet se registra a sí mismo como escucha de eventos del ratón, lo cual se logra al usar **addMouseListener()** y **addMouseMotionListener()**, que son miembros de **Component**. Aquí se muestran:

void addMouseListener (MouseListener *er*)

void addMouseMotionListener (MouseMotionListener *emr*)

En este caso, *er* es una referencia al objeto que recibe los eventos del ratón y *emr* es una referencia al objeto que recibe los eventos de movimiento del ratón. En este programa, se usa el mismo objeto para ambos.

Posteriormente, el applet implementa todos los métodos definidos por las interfaces **MouseListener** y **MouseMotionListener**, que son los manejadores de eventos para los diversos eventos del ratón. Cada método maneja el evento y luego regresa.

HABILIDAD FUNDAMENTAL 14.11

Más palabras clave de Java

Antes de concluir este libro, es necesario analizar brevemente unas cuantas palabras clave más de Java:

- **transient**
- **volatile**
- **instanceof**
- **native**
- **strictfp**
- **assert**

Estas palabras clave se utilizan en programas más avanzados que los que se encuentran en este libro. Sin embargo, se presentará una revisión general de cada una de ellas con el fin de que conozca su objetivo.

Los modificadores transient y volatile

Las palabras clave **transient** y **volatile** son modificadores de tipo que manejan situaciones especializadas. Cuando una variable de instancia se declara como **transient**, entonces no es necesario que persista su valor cuando se almacena un objeto. Por lo tanto, un campo **transient** no afecta, por ejemplo, al estado de un objeto.

El modificador **volatile** se mencionó brevemente en el módulo 11 pero merece un análisis más detallado. La modificación de una variable con **volatile** le indica al compilador que otras partes de su programa pueden cambiar la variable de manera inesperada. Como vio en el módulo 11, una de estas situaciones se relaciona con programas de subprocesos múltiples. En un programa de subprocesos múltiples, a veces dos o más subprocesos compartirán la misma variable de instancia. Por razones de eficiencia, cada subproceso puede mantener su propia copia privada de ese tipo de variable compartida. La copia real (o *maestra*) de la variable se actualiza en varios momentos, como cuando se introduce un método **synchronized**. Aunque este método funciona bien, habrá ocasiones en que resulte inapropiado. En algunos casos, todo lo que interesa es que la copia maestra de una variable siempre refleje el estado actual. Para asegurar esto, simplemente especifique la variable como **volatile**. Al hacerlo así se le indica al compilador que siempre debe mantener actualizada cualquier copia

privada con la copia maestra, y viceversa. Además, deben ejecutarse los accesos a la variable maestra en el orden preciso en que se ejecutan en cualquier copia privada.

instanceof

En ocasiones resulta útil conocer en tiempo de ejecución el tipo de un objeto. Por ejemplo, tal vez tenga un subproceso de ejecución que genere varios tipos de objetos y otro que procese esos objetos. En esta situación, resulta útil para el procesamiento del subproceso conocer el tipo de cada objeto cuando lo recibe. El moldeo una de las situaciones en las que el conocimiento de un tipo de objeto en tiempo de ejecución es importante. En Java, el moldeo no válido causa un error en tiempo de ejecución. Muchos moldeos no válidos pueden capturarse en tiempo de compilación; sin embargo, el moldeo que se relaciona con las jerarquías de clase puede producir moldeos no válidos que sólo pueden detectarse en tiempo de ejecución. Debido a que una referencia a superclase puede hacer referencia a objetos de subclase, no siempre es posible saber en tiempo de compilación si un moldeo que incluye una referencia a una superclase es válido o no. La palabra clave **instanceof** atiende este tipo de situaciones.

El operador **instanceof** tiene esta forma general:

objeto instanceof *tipo*

Aquí, *objeto* es una instancia de una clase y *tipo* es un tipo de clase o interfaz. Si *objeto* es del tipo especificado, o puede moldearse en el tipo especificado, entonces el operador **instanceof** se evalúa como **true**. De otra manera, da como resultado **false**. Por consiguiente, **instanceof** es el medio mediante el cual su programa puede obtener información del tipo de tiempo de ejecución acerca de un objeto.

strictfp

Una de las palabras clave más particulares es **strictfp**. Con la creación de Java 2, el modelo de cálculo de punto flotante se relajó ligeramente para que ciertos cálculos de punto flotante fueran más rápidos para ciertos procesadores, como el Pentium. Específicamente, el nuevo modelo no requiere que se trunquen ciertos valores intermedios que ocurren durante un cálculo. Al modificar una clase o un método con **strictfp**, usted asegura que los cálculos de punto flotante (y, por tanto, todos los truncamientos) tengan lugar precisamente como lo hacían en versiones anteriores de Java. El truncamiento afecta sólo al exponente de ciertas operaciones. Cuando una clase se modifica con **strictfp**, todos los métodos de la clase son también automáticamente **strictfp**.

assert

La palabra clave **assert** se usa durante el desarrollo del programa para crear una *aseveración*, que es una condición que se espera que sea cierta durante la ejecución del programa. Por ejemplo, tal vez tenga un

método que siempre deba regresar un valor entero positivo. Podría probar esto al aseverar que el valor de regreso es mayor que cero usando una instrucción **assert**. En tiempo de ejecución, si la condición es realmente true, no se realizará ninguna otra acción. Sin embargo, si la condición es false, entonces se lanzará un **AssertionError**. Las aseveraciones suelen usarse durante la prueba para verificar que realmente se cumpla alguna condición esperada. Por lo general no se usan para código liberado.

La palabra clave **assert** tiene dos formas. Aquí se muestra la primera:

assert *condición*

Aquí, *condición* es una expresión que debe dar como resultado un valor booleano. Si el resultado es **true**, entonces la aseveración es verdadera y ninguna otra acción se realiza. Si la condición es **false**, entonces la aseveración falla y se lanza un objeto predeterminado **AssertionError**. Por ejemplo.

assert n > 0;

Si **n** es menor o igual a cero, entonces se lanza un **AssertionError**. De otra manera, no se realiza ninguna acción adicional.

Aquí se muestra la segunda forma de **assert**:

assert co*ndición* : *expr*:

En esta versión, *expr* es un valor que se pasa al constructor de **AssertError**. Este valor se convierte en su cadena y se despliega si la aseveración falla. Por lo general, usted especificará una cadena para *expr*, pero se permite cualquier expresión que no sea **void**, siempre y cuando defina una conversión de cadena razonable.

Para permitir la comprobación de una aseveración en tiempo de ejecución, debe especificar la opción **–ea**. Por ejemplo, para permitir aseveraciones para **Muestra**, ejecútela utilizando esta línea:

```
java –ea Muestra
```

Las aseveraciones son muy útiles durante el desarrollo, porque afinan el tipo de comprobación de error que es común durante la prueba. Pero tenga cuidado: no debe depender de una aseveración para realizar cualquier acción realmente requerida por el programa. La razón es que, por lo general, el código liberado se ejecutará con las aseveraciones inhabilitadas y no se evaluará la expresión de una aseveración.

Métodos nativos

Aunque resulte raro, habrá ocasiones en que deseará realizar una llamada a una subrutina que esté escrita en un lenguaje diferente al de Java. Por lo general, este tipo de subrutina existirá como código ejecutable para la CPU y el entorno en que está trabajando (es decir, se tratará de código nativo). Por ejemplo, tal vez desee llamar a una subrutina en código nativo para lograr una ejecución más rápida. O tal vez quiera usar una biblioteca especializada, de terceros, como un paquete de estadísticas. Sin

embargo, debido a que los programas de Java están compilados en código de bytes, que luego es interpretado (o compilado al vuelo) por el sistema en tiempo de ejecución de Java, sería imposible llamar a una subrutina de código nativo desde el interior de su programa en Java. Por fortuna, esta conclusión es falsa. Java proporciona la palabra clave **native**, que se usa para declarar métodos en código nativo. Una vez declarados, estos métodos pueden llamarse desde el interior de su programa de Java de la misma manera en que llama a cualquier otro método de Java.

Para declarar un método nativo, anteceda el método con el modificador **native**, pero no defina ningún cuerpo para el método. Por ejemplo:

```
public native int met() ;
```

Una vez que haya declarado un método nativo, debe escribirlo y seguir una serie más bien compleja de pasos para vincularlo con su código de Java.

¿Qué sucede a continuación?

¡Felicidades! Si ha leído y trabajado los 14 módulos anteriores, entonces ya puede llamarse programador de Java. Por supuesto, aún hay muchas, muchas cosas por aprender acerca de Java, sus bibliotecas y sus subsistemas, pero ahora cuenta con sólidas bases para poder aumentar su conocimiento y su experiencia.

He aquí algunos temas sobre los que tal vez desee aprender más:

- El juego de herramientas abstracto de ventanas (Abstract Window Toolkit, AWT), incluidos sus diversos elementos de interfaz de usuario, como botones para oprimir, menús, listas y barras de desplazamiento.
- Administradores de diseño, que controlan la manera en que se despliegan los elementos en un applet.
- Manejo de salida de texto e imágenes en una ventana.
- El subsistema de manejo de eventos pues, aunque se introdujo aquí, tiene muchos aspectos más.
- Las clases para redes de Java.
- Las clases de utilerías de Java, especialmente su Collections Framework (marco conceptual de colecciones), que simplifica varias tareas comunes de programación.
- La API Concurrent, que ofrece un control detallado sobre aplicaciones de subprocesos múltiples de alto desempeño.
- Swing, que es una alternativa a AWT.
- Java Beans, que soportan la creación de componentes de software en Java.
- Creación de métodos nativos.

Para continuar su estudio de Java, recomiendo el libro *Java: The Complete Reference, J2SE 5 Edition* (McGraw-Hill Osborne, 2005), que cubre todos los temas que se mencionaron, además de muchos más. En él encontrará una cobertura completa del lenguaje Java, sus bibliotecas, subsistemas y aplicaciones.

Comprobación de dominio del módulo 14

1. ¿A qué método se llama cuando un applet empieza a ejecutarse? ¿A qué método se llama cuando se elimina un applet de un sistema?
2. Explique por qué un applet debe usar subprocesos múltiples cuando se requiere ejecutarlo de manera continua.
3. Mejore el proyecto 14.1 para que despliegue la cadena pasada a él como parámetro. Agregue un segundo parámetro que especifique la demora (en milisegundos) entre cada rotación.
4. Desafío adicional: cree un applet que despliegue el tiempo actual y que se actualice una vez por segundo. Para lograrlo, necesitará investigar un poco. He aquí una sugerencia que le ayudará a empezar: la manera más fácil de obtener el tiempo actual es usar un objeto **Calendar**, que es parte del paquete **java.util**. (Recuerde que Sun proporciona documentación para todas las clases estándar de Java.) Ahora ya puede examinar la clase **Calendar** por su cuenta y usar sus métodos para resolver este problema.
5. Explique brevemente el modelo de evento de delegación de Java.
6. ¿Un escucha de eventos debe registrarse a sí mismo con un origen?
7. Desafío adicional: otro de los métodos de despliegue de Java es **drawLine()**. Éste dibuja una línea en el color seleccionado entre dos puntos y es parte de la clase **Graphics**. Empleando **drawLine()**, escriba un programa que registre el movimiento del ratón. Si se oprime el botón, haga que el programa dibuje una línea continua hasta que el botón del ratón se suelte.
8. Describa brevemente la palabra **assert**.
9. Proporcione una razón por la que un método nativo resultaría útil en algunos tipos de programas.
10. Por su cuenta, siga aprendiendo Java. Una buena manera de empezar es el examen de los paquetes estándar de Java, como **java.util**, **java.awt** y **java.net**. Éstos contienen muchas clases que le permitirán crear aplicaciones sorprendentes, listas para Internet.

Apéndice A

Respuestas a las comprobaciones de dominio

Módulo 1: Fundamentos de Java

1. ¿Qué es un código de bytes y por qué es importante su uso en Java para la programación de Internet?

   El código de bytes es un conjunto optimizado de instrucciones que se ejecuta en el intérprete en tiempo de ejecución de Java.
   El código de bytes ayuda a Java a lograr portabilidad y seguridad.

2. ¿Cuáles son los tres principios de la programación orientada a objetos?

   Encapsulamiento, polimorfismo y herencia.

3. ¿Dónde empieza la ejecución de los programas de Java?

   Los programas de Java empiezan su ejecución en **main()**.

4. ¿Qué es una variable?

   Una variable es una ubicación de memoria con nombre. El contenido de una variable puede cambiar durante la ejecución del programa.

5. ¿Cuál de los siguientes nombres de variables no es válido?

   La variable no válida es **D**. Los nombres de una variable no pueden empezar con un dígito.

6. ¿Cómo crea un comentario de una sola línea? ¿Cómo crea uno de varias líneas?

   Un comentario de una sola línea empieza con // y termina al final de una línea. Un comentario de varias líneas empieza con /* y termina con */.

7. Muestre la forma general de la instrucción **if**. Muestre la forma general del bucle **for**.

   La forma general de la instrucción **if**:

   if(*condición*) *instrucción*;

   La forma general de la instrucción **for**:

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

8. ¿Cómo se crea un bloque de código?

   Un bloque de código empieza con un { y termina con un }.

9. La gravedad de la Luna es de alrededor de 17% de la de la Tierra. Escriba un programa que calcule su peso efectivo en la Luna.

```
/*
   Calcula su peso en la Luna.

   Llame a este archivo Luna.java.
*/
```

```
class Luna {
  public static void main(String args[]) {
    double pesotierra; // peso en la Tierra
    double pesoluna; // peso en la Luna

    pesotierra = 74;

    pesoluna = pesotierra * 0.17;

    System.out.println(pesotierra +
                       " kilos en la Tierra equivalen a " +
                       pesoluna + " kilos en la luna.");

  }
}
```

10. Adapte el proyecto 1.2 para que imprima una tabla de conversión de pulgadas a metros. Despliegue 12 pies de conversiones, pulgada por pulgada. Dé salida a una línea en blanco cada 12 pulgadas. (Un metro es aproximadamente igual a 39.37 pulgadas.)

```
/*
   Este programa despliega una tabla de
   conversión de pulgadas a metros.

   Llame a este programa TablaPulgadasAMetros.java.
*/
class TablaPulgadasAMetros {
  public static void main(String args[]) {
    double pulgadas, metros;
    int contador;

    contador = 0;
    for(pulgadas = 1; pulgadas <= 144; pulgadas++) {
      metros = pulgadas / 39.37; // convierta a metros
      System.out.println(pulgadas + " pulgadas son " +
                         metros + " metros.");

      contador++;
      // cada 12 líneas, imprime una línea en blanco
      if(contador == 12) {
        System.out.println();
        contador = 0; // restablece el contador de líneas
      }
    }
  }
}
```

11. Si comete un error de escritura cuando esté ingresando su programa, ¿qué tipo de error aparecerá?

    Un error de sintaxis.

12. ¿Es importante el lugar de la línea en el que coloca una instrucción?

    No, Java es un lenguaje de forma libre.

Módulo 2: Introducción a los tipos de datos y los operadores

1. ¿Por qué Java especifica estrictamente el rango y el comportamiento de sus tipos primitivos?

   Java especifica estrictamente el rango y el comportamiento de sus tipos primitivos para asegurar la portabilidad entre plataformas.

2. ¿Cuál es el tipo de carácter de Java, y en qué difiere del tipo de carácter empleado por muchos otros lenguajes de programación?

   El tipo de carácter de Java es **char**. Los caracteres de Java son Unicode en lugar de ASCII, porque el primero se usa en casi todos los demás lenguajes de cómputo.

3. Un valor **boolean** puede tener cualquier valor que usted desee porque cualquier valor diferente de cero es verdadero. ¿Cierto o falso?

   Falso. Un valor booleano debe ser **true** o **false**.

4. Dada esta salida,

   ```
   Uno
   Dos
   Tres
   ```

   y empleando una sola cadena, muestre la instrucción **println()** que la produce.

   ```
   System.out.println("Uno\nDos\nTres");
   ```

5. ¿Qué está incorrecto en este fragmento?

   ```
   for(i = 0; i < 10; i++) {
     int suma;

     suma = suma + i;
   }
   System.out.println("La suma es: " + suma);
   ```

   Hay dos errores fundamentales en el fragmento. En primer lugar, **suma** se crea cada vez que el bloque creado por el bucle **for** es ingresado y destruidos a la salida. Por lo tanto, no conservará su valor entre iteraciones. No tiene sentido tratar de usar la **suma** para que contenga una suma activa de las iteraciones. En segundo lugar, **suma** no será conocida fuera del bloque en que está declarada. Así, no es válida la referencia a ella en la instrucción **println()**.

6. Explique la diferencia entre las formas de prefijo y sufijo del operador de incremento.

Cuando el operador de incremento antecede a su operando, Java realiza la operación correspondiente antes de obtener el valor del operando para que el resto de la operación lo utilice. Si el operador sigue a su operando, entonces Java obtendrá el valor del operando antes del incremento.

7. Muestra la manera en la que un Y de cortocircuito puede usarse para evitar un error de división entre cero.

```
if((b != 0) && (val / b)) ...
```

8. En una expresión, ¿cuáles tipos son promovidos a **byte** y **short**?

En una expresión, **byte** y **short** se promueven a **int**.

9. En general, ¿cuándo es necesario el moldeado?

Se necesita un moldeado cuando se convierte entre tipos incompatibles o cuando está ocurriendo una conversión de estrechamiento.

10. Escriba un programa que encuentre todos los números primos entre 1 y 100.

```
// Encuentra los números primos entre 1 y 100.
class Primos {
  public static void main(String args[]) {
    int i, j;
    boolean esprimo;

    for(i=1; i < 100; i++) {
      esprimo = true;

      // ve si el número se divide exactamente
      for(j=2; j < i/j; j++)
        // si se divide, entonces no es primo
        if((i%j) == 0) esprimo = false;

      if(esprimo)
        System.out.println(i + " es primo.");
    }
  }
}
```

11. ¿El uso de paréntesis redundantes afecta el desempeño del programa?

No.

12. ¿Un bloque define un alcance?

Sí.

Módulo 3: Instrucciones de control del programa

1. Escriba un programa que lea caracteres del teclado hasta que se reciba un punto. Haga que el programa cuente el número de espacios. Reporte el total al final del programa.

```
// Cuenta espacios.
class Espacios {
  public static void main(String args[])
    throws java.io.IOException {

    char car;
    int espacios = 0;


      System.out.println("Escriba un punto para detener.");

    do {
      car = (char) System.in.read();
      if(car == ' ') espacios++;
    } while(car != '.');

    System.out.println("Espacios: " + espacios);
  }
}
```

2. Muestre la forma general de la escalera **if-else-if**.

if(*condición*)
 instrucción;
else if(*condición*)
 instrucción;
else if(*condición*)
 instrucción;
.
.
.
else
 instrucción;

3. Dado:

```
if(x < 10)
  if(y > 100) {
    if(hecho) x = z;
    else y = z;
  }
else System.out.println("error");  // ¿qué pasa si?
```

¿a qué **if** se asocia el último **else**?

El último **else** se asocia con el **if** externo, que es el **if** más cercano al mismo nivel que el **else**.

4. Muestre la instrucción **for** para un bucle que cuenta de 1000 a 0 de –2 en –2.

```
for(int i = 1000; i >= 0; i -= 2) // ...
```

5. ¿Es válido el siguiente fragmento?

```
for(int i = 0; i < num; i++)
  suma += i;

cuenta = i;
```

No; **i** no es conocido fuera del bucle **for** en que está declarado.

6. Explique lo que hace **break**. Asegúrese de explicar ambas formas.

Un **break** sin una etiqueta causa la terminación inmediata del bucle que lo incluye o de la instrucción switch.
Un **break** con una etiqueta causa que el control se transfiera al final del bloque con etiqueta.

7. En el siguiente fragmento, después de que la instrucción **break** se ejecuta, ¿qué se despliega?

```
for(i = 0; i < 10; i++)  {
  while(corriendo) {
    if(x<y) break;
     // …
  }
  System.out.println("Después de while");
}
System.out.println("Después de for");
```

Después de que **break** se ejecuta, se despliega "Después de while".

8. ¿Qué imprime el siguiente fragmento?

```
for(int i = 0; i<10; i++) {
  System.out.println(i + " ");
  If(i%2) == 0) continue;
  System.out.println();
}
```

He aquí la respuesta:

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

9. La expresión de iteración en un bucle **for** no siempre necesita modificar la variable de control del bucle en una cantidad fija. En cambio, la variable puede cambiar de manera arbitraria. Empleando este concepto, escriba un programa que use un bucle **for** para generar y desplegar la progresión 1, 2, 4, 8, 16, 32, etc.

```
/* Use un bucle for para generar la progresión

   1 2 4 8 16,...
*/
class Progress {
  public static void main(String args[]) {

    for(int i = 1; i < 100; i += i)
      System.out.print(i + " ");
  }
}
```

10. Las letras minúsculas en ASCII están separadas de las mayúsculas por 32 caracteres. Por lo tanto, para convertir una minúscula en mayúscula, se restan 32. Use esta información para escribir un programa que lea caracteres del teclado. Haga que se conviertan todas las minúsculas en mayúsculas, y todas las mayúsculas en minúsculas, desplegando el resultado. No le haga cambios a ningún otro carácter. Haga que el programa se detenga cuando el usuario oprima el punto. Al final, haga que el programa despliegue el número de cambios de letras que se llevó a cabo.

```
// Cambia a mayúsculas.
class CambMay {
  public static void main(String args[])
    throws java.io.IOException {
    char car;
    int cambios = 0;

    System.out.println("Escriba un punto para detener.");

    do {
      car = (char) System.in.read();
      if(car >= 'a' & car <= 'z') {
        car -= 32;
        cambios++;
        System.out.println(car);
      }
      else if(car >= 'A' & car <= 'Z') {
        car += 32;
        cambios++;
        System.out.println(car);
      }
```

```
    } while(car != '.');
    System.out.println("Cambios a mayúsculas: " + cambios);
  }
}
```

11. ¿Qué es un bucle infinito?

Un bucle infinito es un bucle que se ejecuta de manera indefinida.

12. Cuándo utiliza **break** con una etiqueta, ¿la etiqueta debe estar en un bloque que contenga **break**?

Sí.

Módulo 4: Introducción a clases, objetos y métodos

1. ¿Cuál es la diferencia entre una clase y un objeto?

Una clase es una abstracción lógica que describe la forma y el comportamiento de un objeto. Un objeto es una instancia física de una clase.

2. ¿Cómo se define una clase?

Una clase se define usando la palabra clave class. Dentro de la instrucción class, usted especifica el código y los datos que conforman la clase.

3. ¿De qué tiene cada objeto una copia propia?

Cada objeto de una clase tiene su propia copia de las variables de instancia de la clase.

4. Empleando dos instrucciones separadas, muestre cómo declarar un objeto llamado **contador** de una clase llamada **MiContador**.

```
MiContador contador;
Contador = new MiContador();
```

5. Muestre cómo se declara un método llamado **MiMet()** si tiene un tipo de retorno **double** y tiene dos parámetros **int** llamados **a** y **b**.

```
double miMet(int a, int b) { // ...
```

6. ¿Cómo debe regresar un método si regresa un valor?

Un método que devuelve un valor debe regresar por medio de la instrucción return, devolviendo el valor de regreso en el proceso.

7. ¿Qué nombre tiene un constructor?

Un constructor tiene el mismo nombre que su clase.

8. ¿Qué función tiene **new**?

   El operador new asigna memoria a un objeto y lo inicializa empleando el constructor del objeto.

9. ¿Qué es la recolección de basura y cómo funciona? ¿Qué es **finalize()**?

   La recolección de basura es el mecanismo que recicla los objetos no usados para que su memoria pueda volver a usarse. Antes de que se recicle un objeto, se llama al método **finalize()** de un objeto.

10. ¿Qué es **this**?

    La palabra clave this es una referencia al objeto en que se invoca a un método. Se pasa automáticamente a un método.

11. ¿Un constructor puede tener uno o más parámetros?

    Sí.

12. Si un método no regresa un valor, ¿cuál debe ser su tipo de regreso?

    void

Módulo 5: Más tipos de datos y operadores

1. Muestre dos maneras de declarar una matriz de una dimensión de 12 **double**.

```
double x[] = new double[12];
double[] x = new double[12];
```

2. Muestre cómo inicializar una matriz de una dimensión de enteros a los valores del 1 al 5.

```
int x[] = { 1, 2, 3, 4, 5 };
```

3. Escriba un programa que use una matriz para encontrar el promedio de 10 valores **double**. Use los 10 valores que desee.

```
// Promedia 10 valores double.
class Prom {
  public static void main(String args[]) {
    double nums[] = { 1.1, 2.2, 3.3, 4.4, 5.5,
                      6.6, 7.7, 8.8, 9.9, 10.1 };
    double suma = 0;

    for(int i=0; i < nums.length; i++)
      suma += nums[i];

    System.out.println("Promedio: " + suma / nums.length);
  }
}
```

4. Cambie el orden del proyecto 5.1 para que ordene una matriz de cadenas. Demuestre que funciona.

```
// Demuestra el orden de Burbuja con cadenas.
class BurbujaCadenas {
  public static void main(String args[]) {
    String cads[] = {
                     "Es", "una", "prueba", "de",
                     "un", "orden", "con", "cadenas"
                    };
    int a, b;
    String t;
    int dimen;

    dimen = cads.length; // número de elementos por ordenar

    // despliega la matriz original
    System.out.print("La matriz original es:");
    for(int i=0; i < dimen; i++)
      System.out.print(" " + cads[i]);
    System.out.println();

    // Este es el orden de burbuja para cadenas.
    for(a=1; a < dimen; a++)
      for(b=dimen-1; b >= a; b--) {
        if(cads[b-1].compareTo(cads[b]) > 0) { // Si está fuera de orden
          // intercambia elementos
          t = cads[b-1];
          cads[b-1] = cads[b];
          cads[b] = t;
        }
      }

    // despliega matriz ordenada
    System.out.print("La matriz ordenada es:");
    for(int i=0; i < dimen; i++)
      System.out.print(" " + cads[i]);
    System.out.println();
  }
}
```

5. ¿Cuál es la diferencia entre los métodos de **String indexOf()** y **lastIndexOf()**?

El método **indexOf()** encuentra la primera aparición de la subcadena especificada. **lastIndexOf()** encuentra la última aparición.

6. Como todas las cadenas son objetos de tipo **String**, muestre cómo puede llamar a los métodos **lenght()** y **charAt()** en esta cadena literal: “Me gusta Java”.

   Aunque parezca extraño, ésta es una llamada válida de **lenght()**:

   ```
   System.out.println(“Me gusta Java”.lenght());
   ```

   La salida desplegada es 11. Se llama a **charAt()** de manera parecida.

7. Expandiendo la clase de cifrado **Codificar**, modifíquela para que use una cadena de ocho caracteres como clave.

```
// Un cifrado de XO mejorado.
class Codificar {
  public static void main(String args[]) {
    String msj = “Esta es una prueba”;
    String codmsj = “”;
    String decmsj = “”;
    String clave = “abcdefgi”;
    int j;

    System.out.print(“Mensaje original: “);
    System.out.println(msj);

    // codifica el mensaje
    j = 0;
    for(int i=0; i < msj.length(); i++) {
      codmsj = codmsj + (char) (msj.charAt(i) ^ clave.charAt(j));
      j++;
      if(j==8) j = 0;
    }

    System.out.print(“Mensaje codificado: “);
    System.out.println(codmsj);

    // decodifica el mensaje
    j = 0;
    for(int i=0; i < msj.length(); i++) {
      decmsj = decmsj + (char) (codmsj.charAt(i) ^ clave.charAt(j));
      j++;
      if(j==8) j = 0;
    }

    System.out.print(“Mensaje decodificado: “);
    System.out.println(decmsj);

  }
}
```

8. ¿Pueden aplicarse los operadores de bitwise al tipo **double**?

No

9. Muestre cómo puede reescribirse esta secuencia usando el operador **?**

```
if(x < 0) y = 10;
else y = 20;
```

He aquí la respuesta:

```
y = x < 0 ? 10 : 20;
```

10. En el siguiente fragmento, ¿el operador **&** es lógico o de bitwise? ¿Por qué?

```
boolean a, b;
// ...
if(a & b) ...
```

Es un operador lógico porque los operandos son de tipo **boolean**.

11. ¿Es un error sobrepasar el final de una matriz?

Sí.

¿Es un error indizar una matriz con un valor negativo?

Sí. Todos los índices de matriz empiezan en cero.

12. ¿Cuál es el operador de desplazamiento a la derecha sin signo?

>>>

13. Reescriba la clase **MinMax** mostrada antes en este capítulo para que use un bucle **for** de estilo for-each.

```
// Encuentra los valores mínimo y máximo de una matriz.
class MinMax {
  public static void main(String args[]) {
    int nums[] = new int[10];
    int min, max;

    nums[0] = 99;
    nums[1] = -10;
    nums[2] = 100123;
    nums[3] = 18;
    nums[4] = -978;
    nums[5] = 5623;
    nums[6] = 463;
    nums[7] = -9;
    nums[8] = 287;
```

```
    nums[9] = 49;

    min = max = nums[0];
    // Use a for-each style for loop.
    for(int v : nums) {
      if(v < min) min = v;
      if(v > max) max = v;
    }
    System.out.println("Mínimo y máximo: " + min + " " + max);
  }
}
```

14. ¿Es posible convertir en bucles de estilo for-each los **for** que realizan el ordenamiento en la clase **Burbuja** mostrada en el proyecto 5.1? Si no así, ¿por qué?

No, los bucles **for** en la clase **Burbuja** que realiza el ordenamiento no pueden convertirse en bucles de estilo for-each. En el caso del bucle externo, el valor actual de su contador de bucle es necesario para el bucle interno. En el caso del bucle interno, los valores fuera de orden deben intercambiarse, lo que requiere asignaciones. Las asignaciones a la matriz no pueden darse empleando un bucle de estilo for-each.

Módulo 6: Un análisis detallado de métodos y clases

1. Dado este fragmento,

```
class X  {
  private int cuenta;
```

¿el siguiente programa es correcto?

```
class  Y {
  public static void main(String args[]) {
    X ob = new X();

    ob.cuenta = 10;
```

No, un miembro **private** no puede ser accedido fuera de su clase.

2. Un especificador de acceso debe _____ una declaración de miembro.

anteceder

3. El complemento de una cola es una pila: utiliza el acceso primero en entrar, último en salir y suele compararse con una pila de platos. El primer plato puesto en la mesa es el último en usarse. Cree una clase de pila llamada **Pila** que contenga caracteres; llame a los métodos que acceden a la pila **quitar()** y **poner()**; permita al usuario especificar el tamaño de la pila cuando ésta se cree, y mantenga como private a todos los demás miembros de la clase **Pila**. (Sugerencia: puede usar la clase **Cola** como modelo; sólo cambie la manera en la que se tiene acceso a los datos.)

```
// Una clase de pila para caracteres.
class Pila {
  private char pila[]; // esta matriz contiene la pila
  private int parr;  // parte superior de la pila

  // Construye una pila vacía dado su tamaño.
  Pila(int dimen) {
    pila = new char[dimen]; // asigna memoria a la pila
    parr = 0;
  }

  // Construye una pila a partir de Pila.
  Pila(Pila ob) {
    parr = ob.parr;
    pila = new char[ob.pila.length];

    // copia elementos
    for(int i=0; i < parr; i++)
      pila[i] = ob.pila[i];
  }

  // Construye una pila con valores iniciales.
  Pila(char a[]) {
    pila = new char[a.length];

    for(int i = 0; i < a.length; i++) {
      poner(a[i]);
    }
  }


  // coloca caracteres en la pila.
  void colocar(char ch) {
    if(parr==pila.length) {
      System.out.println(" -- La pila se ha llenado.");
      return;
    }

    pila[parr] = ch;
    parr++;
  }

  // Quita un carácter de la pila.
  char quitar() {
    if(parr==0) {
      System.out.println(" -- La pila está vacía.");
```

```
      return (char) 0;
    }

    parr--;
    return pila[parr];
  }
}

// Demuestra la clase Pila.
class PDemo {
  public static void main(String args[]) {
    // construye una pila vacía de 10 elementos
    Pila pila1 = new Pila(10);

    char nombre[] = {'T', 'o', 'm'};

    // construye una pila de la matriz
    Pila pila2 = new Pila(nombre);

    char car;
    int i;

    // coloca algunos caracteres en pila1
    for(i=0; i < 10; i++)
      pila1.poner((char) ('A' + i));

    // construye una pila de otra pila
    Pila pila3 = new Pila(pila1);

    //muestra las pilas.
    System.out.print("Contenido de pila1: ");
    for(i=0; i < 10; i++) {
      car = pila1.quitar();
      System.out.print(car);
    }

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

    System.out.print("Contenido de pila2: ");
    for(i=0; i < 3; i++) {
      car = pila2.quitar();
      System.out.print(car);
    }

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

```
    System.out.print("Contenido de pila3: ");
    for(i=0; i < 10; i++) {
      car = pila3.quitar();
      System.out.print(car);
    }
  }
}
```

He aquí la salida del programa:

```
Contenido de pila1: JIHGFEDCBA

Contenido de pila2: moT

Contenido de pila3: JIHGFEDCBA
```

4. Dada esta clase:

```
class Prueba  {
    int a;
    Prueba(int a) { a = i; }
}
```

escriba un método llamado **cambiar()** que intercambie el contenido de los objetos a que hacen referencia las dos referencias a objeto de **Prueba**.

```
void swap(Prueba ob1, Prueba ob2) {
  int t;

  t = ob1.a;
  ob1.a = ob2.a;
  ob2.a = t;
}
```

5. ¿El siguiente fragmento es correcto?

```
class X  {
  int met(int a, int b) { ... }
  String met(int a, int b) { ... }
```

No. Los métodos sobrecargados pueden tener tipos de regreso diferentes, pero no participan en la resolución de sobrecarga. Los métodos sobrecargados *deben* tener listas de parámetros diferentes.

6. Escriba un método recursivo que despliegue hacia atrás el contenido de una cadena.

```
// Despliega una cadena hacia atrás usando recursión.
class Reversa {
  String cad;

  Reversa(String c) {
```

```
    cad = c;
  }

  void reversa(int ind) {
    if(ind != cad.length()-1) reversa(ind+1);

    System.out.print(cad.charAt(ind));
  }
}

class RVDemo {
  public static void main(String args[]) {
    Reversa c = new Reversa("Es una prueba");

    c.reversa(0);
  }
}
```

7. Si todos los objetos de una clase necesitan compartir la misma variable, ¿cómo debe declarar esa variable?

Las variables compartidas se declaran como **static**.

8. ¿Por qué necesitaría usar un bloque **static**?

Un bloque **static** se usa para realizar cualquier inicialización relacionada con la clase, antes de que cualquier objeto se cree.

9. ¿Qué es una clase interna?

Una clase interna es una clase anidada no estática.

10. Para que un miembro sea accesible únicamente por otro miembro de su clase, ¿qué especificador de acceso debe utilizarse?

private

11. El nombre de un método más su lista de parámetros constituye _____ del método.

una firma

12. Un argumento **int** se pasa a un método usando una llamada por _____.

valor

13. Cree un método varargs llamado **suma()** que sume los valores **int** que se le pasen. Haga que regrese el resultado. Demuestre su uso.

Hay muchas maneras de elaborar la solución. He aquí una:

```
class SumaTotal {
  int suma(int ... n) {
    int resultado = 0;
```

```
    for(int i = 0; i < n.length; i++)
      resultado += n[i];

    return resultado;
  }
}

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

    SumaTotal sumObj = new SumaTotal();

    int total = sumObj.suma(1, 2, 3);
    System.out.println("La suma es " + total);

    total = sumObj.suma(1, 2, 3, 4, 5);
    System.out.println("La suma es " + total);
  }
}
```

14. ¿Puede sobrecargarse un método varargs?

Sí.

15. Muestre un ejemplo de un método varargs sobrecargado que sea ambiguo.

He aquí un ejemplo de un método varargs sobrecargado que es ambiguo:

```
double miMet(double ... v ) { // ...

double miMet(double d, double ... v ) { // ...
```

Si trata de llamar a **miMet()** con un argumento como éste:

```
miMet(1.1);
```

el compilador no podrá determinar qué versión del método invocar.

Módulo 7: Herencia

1. ¿Una superclase tiene acceso a los miembros de una subclase? ¿Una subclase tiene acceso a los miembros de una superclase?

No, una superclase no conoce sus subclases. Sí, una subclase tiene acceso a todos los miembros no privados de su superclase.

2. Cree una subclase de **FormaDosD** llamada **Círculo**. Incluya un método **área()** que calcule el área del círculo y un constructor que use **súper** para inicializar la parte de **FormaDosD**.

```
// Una subclase de FormaDosD para círculos.
class Círculo extends FormaDosD {
  // Un constructor predeterminado.
  Círculo() {
    súper();
  }

  // Construye el círculo
  Círculo(double x) {
    super(x, "círculo"); // Llama al constructor de la superclase
  }

  // Construye un objeto a partir de otro.
  Círculo(Círculo ob) {
    super(ob); // pasa un objeto al constructor de FormasDosD
  }

  double area() {
    return (obteneranchо() / 2) * (obteneranchо() / 2) * 3.1416;
  }
}
```

3. ¿Cómo evita que una subclase tenga acceso a los miembros de una superclase?

Para evitar que una subclase tenga acceso a un miembro de una superclase, declare ese miembro como **private**.

4. Describa el objetivo y el uso de ambas versiones de **súper**.

La palabra clave **súper** tiene dos formas. La primera se usa para llamar a un constructor de una superclase. La forma general de este uso es:

super(*lista-param*);

La segunda forma de súper se usa para acceder a un miembro de una superclase. Esta es su forma general:

súper.*miembro*

5. Dada la siguiente jerarquía:

```
class Alfa { ...

class Beta extends Alfa { ...

class Gamma exteds Beta { ...
```

Los constructores son llamados siempre en orden de derivación. Por lo tanto, cuando se crea un objeto de **Gamma**, el orden es **Alfa**, **Beta**, **Gamma**.

6. Una referencia a una superclase puede hacer referencia a un objeto de una subclase. Explique por qué esto es importante esto en relación con la sobrescritura de métodos.

   Cuando un método sobrescrito se llama mediante una referencia a una superclase, es el tipo de objeto al que se hace referencia el que determina cuál versión del método se llama.

7. ¿Qué es una clase abstracta?

   Una clase abstracta contiene por lo menos un método abstracto.

8. ¿Cómo evita que un método se sobrescriba? ¿Cómo evita que una clase se herede?

   Para evitar que un método se sobrescriba, declárelo como **final**. Para evitar que una clase se herede, declárela como **final**.

9. Explique cómo se emplean la herencia, la sobrescritura de métodos y las clases abstractas para soportar el polimorfismo.

   La herencia, la sobrescritura de métodos y las clases abstractas soportan polimorfismo al permitir que usted cree una estructura de clase generalizada que pueda ser implementada por diversas clases. Por consiguiente, la clase abstracta define una interfaz consistente que es compartida por todas las clases que la implementan. Esto encarna el concepto de "una interfaz, varios métodos".

10. ¿Cuál clase es una superclase de todas las demás clases?

    La clase **Object**.

11. Una clase que contiene por lo menos un método abstracto debe, en sí misma, declararse como abstracta. ¿Cierto o falso?

    Cierto.

12. ¿Cuáles palabras clave se usan para crear una constante con nombre?

    final

Módulo 8: Paquetes e interfaces

1. Mediante el código del proyecto 8.1, coloque la interfaz **ICCar.java** y sus tres implementaciones en un paquete llamado **PaqueteC**. Manteniendo la demostración de la cola de la clase **ICDemo** en el paquete predeterminado, muestre cómo importar y usar las clases de **PaqueteC**.

   Para colocar **ICCar** y sus implementaciones en el paquete **PaqueteC**, debe separar cada una en su propio archivo, hacer **public** cada implementación de clase y agregar esta instrucción al principio de cada archivo.

```
package PaqueteC;
```

Una vez que se ha realizado esto, puede usar **PaqueteC** al agregar esta instrucción **import** a **ICDemo**.

```
import PaqueteC.*;
```

2. ¿Qué es un espacio de nombre? ¿Por qué es importante que Java le permita dividir el espacio de nombre?

   Un espacio de nombre es una región declarativa. Al fragmentar ese espacio de nombre, puede evitar colisiones entre nombres.

3. Los paquetes están almacenados en _____.

   directorios

4. Explique la diferencia entre el acceso **protected** y el predeterminado.

   Un miembro con acceso **protected** puede usarse dentro de su paquete y una subclase puede usarlo en cualquier paquete. Un miembro con acceso predeterminado sólo puede usarse dentro de su paquete.

5. Explique las dos maneras en que otros paquetes pueden usar a los miembros de un paquete.

   Para usar un miembro de un paquete, puede calificar por completo su nombre, e importarlo usando **import**.

6. "Una interfaz, varios métodos" es un concepto clave de Java. ¿Qué característica lo ejemplifica mejor?

   La interfaz es la característica que ejemplifica mejor el principio "una interfaz, varios métodos" de la programación orientada a objetos.

7. ¿Cuántas clases pueden implementar una interfaz? ¿Cuántas interfaces puede implementar una clase?

   Una cantidad ilimitada de clases puede implementar una interfaz. Una clase puede implementar todas las interfaces que elija.

8. ¿Es posible extender las interfaces?

   Sí, es posible extender las interfaces.

9. Cree una interfaz para la clase **Automotor** del módulo 7. Llámela **IAutomotor**.

```
interface IAutomotor {

  // Regresa el rango.
  int rango();

  // Calcula la gasolina para una distancia dada.
  double gasnecesaria(int km);

  // Métodos de acceso para variables de instancia.
  int obtenerPasajeros();
  void establcerPasajeros(int p);
  int obtenerTanquegas();
```

```
  void establecerTanquegas(int f);
  int obtenerKpl();
  void establecerKpl(int m);
}
```

10. Las variables declaradas en una interfaz son implícitamente **static** y **final**. ¿Para qué sirven?

Las variables de interfaz son valiosas como constantes con nombre que son compartidas por todos los archivos de un programa. Se traen a la vista al importar su interfaz.

11. En esencia, un paquete es un contendedor de clases. ¿Cierto o falso?

Cierto.

12. ¿Qué paquete estándar de Java se importa automáticamente en un programa?

java.lang

Módulo 9: Manejo de excepciones

1. ¿Cuál es la clase superior en la jerarquía de excepciones?

Throwable se encuentra en la parte superior de la jerarquía de excepciones.

2. Explique brevemente cómo usar **try** y **catch**.

Las instrucciones **try** y **catch** trabajan juntas. Las instrucciones del programa que usted desea monitorear en busca de excepciones están contenidas dentro del bloque **try**. Una excepción se captura usando **catch**.

3. ¿Qué está incorrecto en este fragmento?

```
// ...
vals[18] = 10;
catch (ArrayIndexOutOfBoundsException exc) {
  // maneja error
}
```

No hay un bloque **try** antes de la instrucción **catch**.

4. ¿Qué pasa si no se captura una excepción?

Si no se captura una excepción, se obtiene la terminación anormal del programa.

5. ¿Qué está incorrecto en este fragmento?

```
class A extends Exception { ...

class B extends A { ...

// ...
```

```
try {
  // ...
}
catch (A exc) { ... }
catch (B exc) { ... }
```

En el fragmento, la **catch** de una superclase antecede a la de una subclase. Debido a que la **catch** de la superclase capturará también todas las subclases, se crea código que no es posible alcanzar.

6. ¿Una excepción capturada por un bloque de **catch** interno puede relanzar esa excepción a un bloque de **catch** externo?

   Sí, es posible volver a lanzar una excepción.

7. El bloque de **finally** es el último fragmento de código ejecutado antes de que su programa termine. ¿Cierto o falso?

   Falso. El bloque de **finally** es el código ejecutado cuando termina un bloque de **try**.

8. ¿Qué tipo de excepción debe declararse explícitamente en una cláusula **throw** de un método?

   Todas las excepciones, menos las de tipo **RuntimeException** y **Error**, deben declararse en una cláusula **throws**.

9. ¿Qué está incorrecto en este fragmento?

```
class MiClase { // ... }
// ...
throw new MiClase();
```

   MiClase no extiende **Throwable**. Sólo las subclases de **Throwable** pueden ser lanzadas con **throw**.

10. En la pregunta 3 de la Comprobación de dominio del módulo 6, creó una clase **Pila**. Agregue excepciones personalizadas a su clase que reporten las condiciones de pila llena y pila vacía.

```
// Una excepción para errores de pila llena.
class ExcepciónPilaLlena extends Exception {
  int dimen;

  ExcepciónPilaLlena(int s) { dimen = s; }

  public String toString() {
   return "\nSLa pila está llena. El tamaño máximo es " +
          dimen;
  }
}

// Una excepción para errores de pila vacía.
class ExcepciónPilaVacía extends Exception {

  public String toString() {
```

```
    return "\nLa pila está vacía.";
  }
}

// Una clase de pila para caracteres.
class Pila {
  private char pila[]; // esta matriz contiene la pila
  private int parr;  // parte superior de la pila

  // Construye una pila vacía dado su tamaño.
  Pila(int dimen) {
    pila = new char[dimen]; // asigna memoria a la pila
    parr = 0;
  }

  // Construye una pila a partir de Pila.
  Pila(Pila ob) {
    parr = ob.parr;
    pila = new char[ob.pila.length];

    // copia elementos
    for(int i=0; i < parr; i++)
      pila[i] = ob.pila[i];
  }

  // Construye una pila con valores iniciales.
  Pila(char a[]) {
    pila = new char[a.length];

    for(int i = 0; i < a.length; i++) {
      try {
        poner(a[i]);
      }
      catch(ExcepciónPilaLlena exc) {
        System.out.println(exc);
      }
    }
  }

  // Pone caracteres en la pila.
  void poner(char car) throws ExcepciónPilaLlena {
    if(parr==pila.length)
      throw new ExcepciónPilaLlena(pila.length);

    pila[parr] = car;
    parr++;
```

```
  }

  // Quita un carácter de la pila.
  char quitar() throws ExcepciónPilaVacía {
    if(parr==0)
      throw new ExcepciónPilaVacía();

    parr--;
    return pila[parr];
  }
}
```

11. ¿Cuáles son las tres maneras en las que puede una excepción generarse?

    Puede generarse una excepción por un error en la JVM, un error en su programa, o explícitamente mediante una instrucción **throw**.

12. ¿Cuáles son las dos subclases directas de **Throwable**?

    Error y **Exception**.

Módulo 10: Uso de E/S

1. ¿Por qué Java define flujos de bytes y caracteres?

   Los flujos de bytes son los flujos originales definidos por Java. Resultan especialmente útiles para E/S binaria y soportan archivos de acceso directo. Los flujos de caracteres están optimizados para Unicode.

2. Si bien la entrada y la salida de la consola son de texto, ¿por qué Java usa aún flujos de bytes para este fin?

   Los flujos predefinidos, **System.in** y **System.err,** se definieron antes de que Java agregara los flujos de caracteres.

3. Muestre cómo abrir un archivo para lectura de bytes.

   He aquí una manera de abrir un archivo para entrada de **byte**:

   ```
   FileInputStream fin = new FileInputStrem("prueba");
   ```

4. Muestre cómo abrir un archivo para lectura de caracteres.

   He aquí la manera de abrir un archivo para lectura de caracteres:

   ```
   FileReader formulario = new FileReader("prueba");
   ```

5. Muestre cómo abrir un archivo para E/S de acceso directo.

   He aquí una manera de abrir un archivo para acceso directo:

   ```
   archivodirecto = new RandomAccessFile("prueba", "rw");
   ```

6. ¿Cómo puede convertir una cadena numérica como "123.23" en su equivalente binario?

Para convertir cadenas numéricas en sus equivalentes binarios, use los métodos de análisis definidos por los envolventes de tipo, como **Integer** o **Double**.

7. Escriba un programa que copie un archivo de texto. En el proceso, haga que convierta todos los espacios en guiones. Use las clases de archivo de flujo de bytes.

```
/* Copia un archivo de texto, sustituyendo espacios con guiones.

   Esta versión usa flujos de bytes.

   Para usar este programa, especifique el nombre
   de los archivos de origen y de destino.
   Por ejemplo,

   java Guiones origen destino
*/

import java.io.*;

class Guiones {
  public static void main(String args[])
    throws IOException
  {
    int i;
    FileInputStream aren;
    FileOutputStream arsal;

    try {
      // abre archivo de entrada
      try {
        fin = new FileInputStream(args[0]);
      } catch(FileNotFoundException exc) {
        System.out.println("Archivo de entrada no encontrado");
        return;
      }

      // abre archivo de salida
      try {
        asal = new FileOutputStream(args[1]);
      } catch(FileNotFoundException exc) {
        System.out.println("Error al abrir el archivo de salida");
        return;
      }
    } catch(ArrayIndexOutOfBoundsException exc) {
      System.out.println("Uso: Guiones De A");
```

```
      return;
    }

    // Copia archivo
    try {
      do {
        i = fin.read();
        if((char)i == ' ') i = '-';
        if(i != -1) asal.write(i);
      } while(i != -1);
    } catch(IOException exc) {
      System.out.println("Error de archivo");
    }

    fin.close();
    fout.close();
  }
}
```

8. Reescriba el programa descrito en la pregunta 7 para que use las clases de flujo de caracteres.

```
/* Copia un archivo de texto, sustituyendo guiones con espacios.

   Esta versión usa flujos de caracteres.

   Para usar este programa, especifique el nombre
   de los archivos de origen y de destino.
   Por ejemplo,

   java Guiones2 origen destino
*/

import java.io.*;

class Guiones2 {
  public static void main(String args[])
    throws IOException
  {
    int i;
    FileReader fin;
    FileWriter fout;

    try {
      // abre archivo de entrada
      try {
        fin = new FileReader(args[0]);
```

```
      } catch(FileNotFoundException exc) {
        System.out.println("Archivo de entrada no encontrado");
        return;
      }

      // abre archivo de salida
      try {
        fout = new FileWriter(args[1]);
      } catch(IOException exc) {
        System.out.println("Error al abrir el archivo de salida");
        return;
      }
    } catch(ArrayIndexOutOfBoundsException exc) {
      System.out.println("Uso: Guiones2 De A");
      return;
    }

    // Copia archivo
    try {
      do {
        i = fin.read();
        if((char)i == ' ') i = '-';
        if(i != -1) fout.write(i);
      } while(i != -1);
    } catch(IOException exc) {
      System.out.println("File Error");
    }

    fin.close();
    fout.close();
  }
}
```

9. ¿Qué clase de flujo es **System.in**?

InputStream

10. ¿Qué regresa el método **read()** de **InputStream** cuando se alcanza el final del flujo?

–1

11. ¿Qué tipo de flujo se usa para leer datos binarios?

DataInputStream

12. **Reader** y **Writer** son la parte superior de la jerarquía de clases de_____?

E/S de caracteres.

Módulo 11: Programación con varios subprocesos

1. ¿Por qué la capacidad de los subprocesos múltiples de Java le permite escribir programas más eficientes?

   Los subprocesos múltiples le permiten aprovechar el tiempo muerto presente en casi todos los programas. La esencia de los subprocesos múltiples es que cuando no se puede ejecutar un subproceso, se ejecuta otro.

2. Los subprocesos múltiples están soportados por la clase _____ y la interfaz _____.

   Los subprocesos múltiples están soportados por la clase **Thread** y la interfaz **Runnable**.

3. Cuando crea un objeto ejecutable, ¿por qué podría desear extender **Thread** en lugar de implementar **Runnable**?

   Extenderá **Thread** cuando quiera sobrescribir uno o más métodos de **Thread** diferentes de **run()**.

4. Muestre cómo usar **join()** para esperar a que un objeto de subproceso llamado **MiSubpr** termine.

   ```
   MiSubpr.join();
   ```

5. Muestre cómo establecer un subproceso llamado **MiSubpr** a tres niveles por arriba de la prioridad normal.

   ```
   MiSubpr.setpriority(Thread.NORM_PRIORITY+3);
   ```

6. ¿Qué efecto se obtiene al agregar la palabra clave **synchronized** a un método?

   La adición de **synchronized** a un método sólo permite que un subproceso a la vez use el método para cualquier objeto dado de su clase.

7. Los métodos **wait()** y **notify()** se usan para realizar _____.

   comunicación entre subprocesos.

8. Cambie la clase **Tictac** para que realmente lleve el tiempo. Es decir, haga que cada tic tome medio segundo y que cada tac tome otro medio segundo. De esta manera, cada tictac tomará un segundo. (No se preocupe por el tiempo que las tareas de cambio tomen.)

   Para hacer que la clase **Tictac** realmente lleve el tiempo simplemente agreue llamar a **sleep** como se muestra.

   ```
   // Hace que la clase Tictac realmente lleve el tiempo.

   class Tictac {

     synchronized void tic(boolean ejecutando) {
       if(!ejecutando) { // detiene el reloj
         notify(); // subprocesos de notificación y espera
         return;
       }

       System.out.print("Tic ");
   ```

```
    // espera 1/2 segundo
    try {
      Thread.sleep(500);
    } catch(InterruptedException exc) {
      System.out.println("Subproceso interrumpido.");
    }

    notify(); // que tac() se ejecute
    try {
      wait(); // espera a que se complete tac
    }
    catch(InterruptedException exc) {
      System.out.println("Subproceso interrumpido.");
    }
  }

  synchronized void tac(boolean ejecutando) {
    if(!ejecutando) { // detiene el reloj
      notify(); // subprocesos de notificación y espera
      return;
    }

    System.out.println("Tac");

    // espera 1/2 segundo
    try {
      Thread.sleep(500);
    } catch(InterruptedException exc) {
      System.out.println("Subproceso interrumpido.");
    }

    notify(); // que se ejecute tic()
    try {
      wait(); // espera a que se complete tic
    }
    catch(InterruptedException exc) {
      System.out.println("Subproceso interrumpido.");
    }
  }
}
```

9. ¿Por qué no puede usar **suspend()**, **resume()** y **stop()** en nuevos programas?

Los métodos **suspend()**, **resume()** y **stop()** se han desautorizado porque pueden causar problemas serios en tiempo de ejecución.

10. ¿Qué método definido por **Thread** obtiene el nombre de un subproceso?

    getName()

11. ¿Qué regresa **isAlive()**?

    Regresa **true** si el subproceso que invoca aún se está ejecutando, y **false** si ha terminado.

Módulo 12: Enumeraciones, autoencuadre e importación de miembros estáticos

1. Se dice que las constantes de enumeración son de *tipo propio*. ¿Qué significa esto?

   En el término *tipo propio*, el "propio" alude al tipo de la enumeración en que está definida la constante. Por lo tanto, una constante de enumeración es un objeto de la enumeración de la que es parte.

2. ¿Qué clases heredan automáticamente todas las enumeraciones?

   La clase **Enum** es heredada automáticamente por todas las enumeraciones.

3. Dada la siguiente enumeración, escriba un programa que use **values()** para mostrar una lista de constantes y sus valores ordinales.

```
enum Herramientas {
  DESARMADOR,  LLAVE, MARTILLO, PINZAS
}
```

   La solución es:

```
enum Herramientas {
  DESARMADOR, LLAVE, MARTILLO, PINZAS
}

class MostrarEnum {
  public static void main(String args[]) {
    for(Herramientas d : Herramientas.values())
      System.out.print(d + " tiene valor ordinal de " +
                       d.ordinal() + '\n');

  }
}
```

4. La simulación del semáforo que se desarrolló en el proyecto 12.1 puede mejorarse con unos cuantos cambios que aprovechen las funciones de la clase de enumeración. En la versión mostrada, la duración de cada color estuvo controlada por la clase **SimuladorSemáforo** al codificar estos valores en el método **run()**. Cambie esto último para que la duración de cada color se almacene en las constantes de la enumeración **ColorSemáforo**. Para ello, necesitará agregar un constructor,

una variable de instancia privada y un método llamado **obtenerDemora( )**. Después de realizar estos cambios, ¿qué mejoras observa? Por su cuenta, ¿puede pensar en otras mejoras? (Sugerencia: trate de usar valores ordinales para cambiar los colores de las luces, en lugar de depender de la instrucción **switch**.)

Aquí se muestra la versión mejorada de la simulación del semáforo. Hay dos mejoras importantes: en primer lugar, una demora de la luz está vinculada ahora con su valor de enumeración, que brinda más estructura al código. En segundo lugar, el método **run( )** ya no necesita usar una instrucción **switch** para determinar la longitud de la demora. En cambio, se pasa **cs.obtenerDemora( )** a **sleep( )**, lo que origina que la demora asociada con el color actual se use automáticamente.

```
// Una versión mejorada de la simulación del semáforo que
// almacena la demora de la luz en ColorSemáforo.

// Una enumeración de los colores de un semáforo.
enum ColorSemáforo {
  ROJO(12000), VERDE(10000), AMARILLO(2000);

  private int demora;

  ColorSemáforo(int d) {
    demora = d;
  }

  int obtenerDemora() { return demora; }
}

// Un semáforo computarizado.
class SimuladorSemáforo implements Runnable {
  private Thread subpr; // contiene el subproceso que ejecuta la simulación
  private ColorSemáforo cs; // contiene el color real del semáforo
  boolean alto = false; // se pone en true para detener la simulación

  SimuladorSemáforo(ColorSemáforo init) {
    cs = init;

    subpr = new Thread(this);
    subpr.start();
  }

  SimuladorSemáforo() {
    cs = ColorSemáforo.ROJO;

    subpr = new Thread(this);
    subpr.start();
  }
```

```
// Inicia la luz.
public void run() {
  while(!alto) {

    // ¡Observe cómo se ha simplificado este código!
    try {
      Thread.sleep(cs.obtenerDemora());
    } catch(InterruptedException exc) {
      System.out.println(exc);
    }

    cambiarColor();
  }
}

// Cambia el color.
synchronized void cambiarColor() {
  switch(cs) {
    case ROJO:
      cs = ColorSemáforo.VERDE;
      break;
    case AMARILLO:
      cs = ColorSemáforo.ROJO;
      break;
    case VERDE:
     cs = ColorSemáforo.AMARILLO;
  }

  notify(); // indica que la luz ha cambiado
}


// espera hasta que ocurre un cambio de luz.
synchronized void esperaCambio() {
  try {
    wait(); // espera el cambio de luz
  } catch(InterruptedException exc) {
    System.out.println(exc);
  }
}

// Regresa el color real.
ColorSemáforo obtenerColor() {
  return cs;
}

// Detiene el semáforo.
void cancel() {
```

```
    alto = true;
  }
}

class SemáforoDemo {
  public static void main(String args[]) {
    SimuladorSemáforo tl = new SimuladorSemáforo(ColorSemáforo.VERDE);

    for(int i=0; i < 9; i++) {
      System.out.println(tl.obtenerColor());
      tl.esperaCambio();
    }

    tl.cancel();
  }
}
```

5. Defina el encuadre y el desencuadre. ¿Cómo afecta el autoencuadre/desencuadre estas acciones?

Encuadre es el proceso de almacenar un valor primitivo en un objeto de envolvente de tipo. Desencuadre es el proceso de recuperar el valor primitivo del envolvente de tipo. El autoencuadre encuadra automáticamente un valor primitivo sin tener que construir explícitamente un objeto. El autodesencuadre recupera automáticamente el valor primitivo de un envolvente de tipo sin tener que llamar explícitamente a un método, como **intValue()**.

6. Cambie el siguiente fragmento a fin de que éste use el autoencuadre.

```
Short val = new Short(123);
```

La solución es:

```
Short val = 123;
```

7. En sus propias palabras, ¿qué hace la importación de miembros estáticos?

La importación de miembros estáticos trae al espacio de nombre global los miembros estáticos de una clase o interfaz. Esto significa que los miembros estáticos pueden usarse sin tener que ser calificados con su nombre de clase o int.

8. ¿Qué lleva a cabo esta instrucción?

```
import static java.lang.Integer.parseInt;
```

La instrucción trae al espacio de nombre global el método **parseInt()** del envolvente de tipo **Integer**.

9. ¿La importación de miembros estáticos está diseñada para situaciones especiales, o es una buena práctica traer a la vista todos los miembros estáticos de todas las clases?

La importación de miembros estáticos está diseñada para casos especiales. Traer a la vista muchos miembros estáticos ocasionará colisiones en el espacio de nombre y desestructurará su código.

10. Una anotación está sintácticamente basada en _____.

    una interfaz.

11. ¿Qué es una anotación de marcador?

    Una anotación de marcador es aquella que no toma argumentos.

12. Una anotación puede aplicarse sólo a métodos. ¿Cierto o falso?

    Falso. Cualquier tipo de declaración puede tener una anotación.

Módulo 13: Elementos genéricos

1. Los elementos genéricos construyen una adición importante a Java porque permiten la creación de código:

   a) Con seguridad de tipo

   b) Reciclable

   c) Confiable

   d) Todo lo anterior

   d) Todo lo anterior

2. ¿Puede usarse un tipo primitivo como un argumento de tipo?

   No, los argumentos de tipo deben ser tipos de objeto.

3. Muestre cómo declarar una clase llamada **CalendarioVuelo** que tome dos parámetros genéricos.

   La solución es

   ```
   class CalendarioVuelo<T, V> {
   ```

4. Partiendo de su respuesta a la pregunta 3, cambie el segundo parámetro de tipo de **CalendarioVuelo** para que pueda extender **Thread**.

   ```
   La solución es
   class CalendarioVuelo<T, V extends Thread> {
   ```

5. Ahora, cambie **CalendarioVuelo** para que su segundo parámetro de tipo sea una subclase de su primer parámetro de tipo.

   La solución es

   ```
   class CalendarioVuelo<T, V extends T> {
   ```

6. En relación con los elementos genéricos, ¿qué es **?** y qué hace?

   El **?** es el argumento de comodín. Coincide con cualquier tipo válido.

7. ¿Es posible limitar el argumento del comodín?

Sí, un comodín puede tener un límite superior o uno inferior.

8. Un método genérico llamado **MiGen()** tiene un parámetro de tipo. Más aún, **MiGen()** tiene un parámetro cuyo tipo es el del parámetro de tipo. También devuelve un objeto de ese parámetro de tipo. Muestre cómo declarar **MiGen()**.

La solución es

```
<T> T MiGen(T o) { // ...
```

9. Dada esta interfaz genérica

```
interface IfIGen<T, extends T> { // ...
```

muestre la declaración de una clase llamada **MiClase** que implementa **IFIGen**.

La solución es

```
class MiClase<T, V extends T> implements IFIGen<T, V> { // ...
```

10. Dada una clase genérica llamada **Contador<T>**, muestre cómo crear un objeto de su tipo bruto.

Para obtener el tipo bruto de **Contador<T>**, simplemente use su nombre sin ningún tipo de especificación, como se muestra aquí:

```
Contador x = new Contador;
```

11. ¿Los parámetros de tipo existen en el tiempo de ejecución?

No. Todos los parámetros de tipo se borran durante la compilación y se sustituyen los modelados apropiados. A este proceso se le denomina borrado.

12. Convierta su solución a la pregunta 10 de la Comprobación de dominio del módulo 9 para que sea genérica. En el proceso, cree una interfaz de pila llamada **PilaIGen** que defina genéricamente las operaciones **poner()** y **quitar()**.

```
// Una pila genérica.

interface PilaIGen<T> {
  void poner(T obj) throws ExcepciónPilaLlena;
  T quitar() throws ExcepciónPilaVacía;
}

// Una excepción para errores de pila llena.
class ExcepciónPilaLlena extends Exception {
  int dimen;

  ExcepciónPilaLlena(int s) { dimen = s; }

  public String toString() {
   return "\nSLa pila está llena. El tamaño máximo es " +
```

```
            dimen;
  }
}

// Una excepción para errores de pila vacía.
class ExcepciónPilaVacía extends Exception {

  public String toString() {
   return "\nLa pila está vacía.";
  }
}

// Una clase de pila para caracteres.
class PilaGen<T> implements PilaIGen<T> {
  private T pila[]; // esta matriz contiene la pila
  private int parr;  // parte superior de la pila

  // Construye una pila vacía dado su tamaño.
  PilaGen(T[] matrizPila) {
    pila = matrizPila;
    parr = 0;
  }

  // Construye una pila a partir de una pila.
  PilaGen(T[] matrizPila, PilaGen<T> ob) {

    parr = ob.parr;
    pila = matrizPila;

    try {
      if(pila.length < ob.pila.length)
        throw new ExcepciónPilaLlena(pila.length);
    }
    catch(ExcepciónPilaLlena exc) {
      System.out.println(exc);
    }

    // Copia elementos.
    for(int i=0; i < parr; i++)
      pila[i] = ob.pila[i];
  }

  // Construye una pila con valores iniciales.
  PilaGen(T[] matrizPila, T[] a) {
    pila = matrizPila;
```

```
    for(int i = 0; i < a.length; i++) {
      try {
        poner(a[i]);
      }
      catch(ExcepciónPilaLlena exc) {
        System.out.println(exc);
      }
    }
  }

  // Pone objetos en la pila.
  public void poner(T obj) throws ExcepciónPilaLlena {
    if(parr==pila.length)
      throw new ExcepciónPilaLlena(pila.length);

    pila[parr] = obj;
    parr++;
  }

  // Quita un objeto de la pila.
  public T quitar() throws ExcepciónPilaVacía {
    if(parr==0)
      throw new ExcepciónPilaVacía();

    parr--;
    return pila[parr];
  }
}

// Demuestra la clase Pila.
class PilaGenDemo {
  public static void main(String args[]) {
    // Construye una pila vacía de 10 elementos enteros.
    Integer iAlmacena[] = new Integer[10];
    PilaGen<Integer> pila1 = new PilaGen<Integer>(iAlmacena);

    // Construye pila de una matriz.
    String nombre[] = {"Uno", "Dos", "Tres"};
    String cadAlmacén[] = new String[3];
    PilaGen<String> pila2 =
        new PilaGen<String>(cadAlmacén, nombre);

    String cad;
    int n;

    try {
```

```
      // Pone algunos valores en pila1.
      for(int i=0; i < 10; i++)
        pila1.poner(i);
    } catch(ExcepciónPilaLlena exc) {
      System.out.println(exc);
    }

    // Construye pila de otra pila.
    String cadAlmacén2[] = new String[3];
    PilaGen<String> pila3 =
        new PilaGen<String>(cadAlmacén2, pila2);

    try {
      // Muestra las pilas.
      System.out.print("Contenido de pila1: ");
      for(int i=0; i < 10; i++) {
        n = pila1.quitar();
        System.out.print(n + " ");
      }

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

      System.out.print("Contenido de pila2: ");
      for(int i=0; i < 3; i++) {
        cad = pila2.quitar();
        System.out.print(cad + " ");
      }

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

      System.out.print("Contentido de pila3: ");
      for(int i=0; i < 3; i++) {
        cad = pila3.quitar();
        System.out.print(cad + " ");
      }

    } catch(ExcepciónPilaVacía exc) {
      System.out.println(exc);
    }

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

Módulo 14: Applets, eventos y temas diversos

1. ¿A qué método se llama cuando un applet empieza a ejecutarse? ¿A qué método se llama cuando se elimina un applet de un sistema?

Cuando empieza un applet, el primer método al que se llama es **init( )**. Cuando se elimina un applet, se llama a **destroy( )**.

2. Explique por qué un applet debe usar subprocesos múltiples cuando se requiere usarlo de manera continua.

Un applet debe usar subprocesos múltiples cuando se requiere usarlo continuamente porque los applets son programas orientados a eventos que no deben entrar en un "modo" de operación. Por ejemplo, si **start( )** nunca regresa, entonces nunca se llamará a **paint( )**.

3. Mejore el proyecto 14.1 para que despliegue la cadena pasada a él como parámetro. Agregue un segundo parámetro que especifique la demora (en milisegundos) entre cada rotación.

```
/* Un applet de letrero simple que usa parámetros.

*/
import java.awt.*;
import java.applet.*;
/*
<applet code="LetreroParam" width=300 height=50>
<param name=mensaje value=" ¡Me gusta Java! ">
<param name=demora value=500>
</applet>
*/

public class LetreroParam extends Applet implements Runnable {
  String msj;
  int demora;
  Thread t;
  boolean marcaAlto;

  // Inicializa t en null.
  public void init() {
    String temp;

    msj = getParameter("mensaje");
    if(msj == null) msj = " Java gobierna Web ";

    temp = getParameter("demora");
    try {
      if(temp != null)
        demora = Integer.parseInt(temp);
      else
```

```
            demora = 250; // predeterminado si no se especifica
        } catch(NumberFormatException exc) {
            demora = 250 ; // predeterminado si está en error
        }

        t = null;
    }

    // Inicia subproceso
    public void start() {
        t = new Thread(this);
        marcaAlto = false;
        t.start();
    }

    // Punto de entrada al subproceso que rige el Letrero.
    public void run() {
        char car;

        // Despliega el letrero
        for( ; ; ) {
            try {
                repaint();
                Thread.sleep(demora);
                car = msj.charAt(0);
                msj = msj.substring(1, msj.length());
                msj += car;
                if(marcaAlto)
                    break;
            } catch(InterruptedException exc) {}
        }
    }

    // Pone en pausa el letrero.
    public void stop() {
        marcaAlto = true;
        t = null;
    }

    // Despliega el letrero.
    public void paint(Graphics g) {
        g.drawString(msj, 50, 30);
    }
}
```

4. Desafío adicional: cree un applet que despliegue el tiempo actual y que se actualice una vez por segundo. Para lograrlo, necesitará investigar un poco. He aquí una sugerencia que le ayudará a empezar: la manera más fácil de obtener el tiempo actual es usar un objeto **Calendar**, que es parte del paquete **java.util**. (Recuerde que Sun proporciona documentación para todas las clases estándar de Java.) Ahora ya debe encontrarse en el punto en que puede examinar la clase **Calendar** por su cuenta y usar sus métodos para resolver este problema.

```
// Un applet de reloj simple.

import java.útil.*;
import java.awt.*;
import java.applet.*;
/*
<object code="Reloj" width=200 height=50>
</object>
*/

public class Reloj extends Applet implements Runnable {
  String msj;
  Thread t;
  Calendar reloj;

  boolean marcaAlto;

  // Inicializa
  public void init() {
    t = null;
    msj = "";
  }

  // Inicia el subproceso
  public void start() {
    t = new Thread(this);
    marcaAlto = false;
    t.start();
  }

  // Punto de entrada para el reloj.
  public void run() {
    // Despliega el reloj
    for( ; ; ) {
      try {
        reloj = Calendar.getInstance();
        msj = "La hora actual es " +
              Integer.toString(reloj.get(Calendar.HOUR));
```

```
        msj = msj + ":" +
              Integer.toString(reloj.get(Calendar.MINUTE));
        msj = msj + ":" +
              Integer.toString(reloj.get(Calendar.SECOND));
        repaint();
        Thread.sleep(1000);
        if(marcaAlto)
          break;
      } catch(InterruptedException exc) {}
    }
  }

  // Pone en pausa el reloj.
  public void stop() {
    marcaAlto = true;
    t = null;
  }

  // Despliega el reloj.
  public void paint(Graphics g) {
    g.drawString(msj, 30, 30);
  }
}
```

5. Explique brevemente el modelo de evento de delegación de Java.

En el modelo de evento de delegación, un *origen* genera un evento y lo envía a uno o más *escuchas*. Un escucha simplemente espera hasta que recibe un evento. Una vez recibido, el escucha procesa el evento y luego regresa.

6. ¿Un escucha de eventos debe registrarse a sí mismo con un origen?

Sí, un escucha debe registrarse con un origen para recibir eventos.

7. Desafío adicional: otro de los métodos de despliegue de Java es **drawLine()**. Éste dibuja una línea en el color seleccionado entre dos puntos y es parte de la clase **Graphics**. Empleando **drawLine()**, escriba un programa que registre el movimiento del ratón. Si se oprime el botón, haga que el programa dibuje una línea continua hasta que el botón del ratón se suelte.

```
/* Rastrea el movimiento del ratón dibujando una línea
   cuando se oprime un botón del ratón. */

import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="RastreoR" width=300 height=100>
```

```
  </applet>
*/

public class RastreoR extends Applet
  implements MouseListener, MouseMotionListener {

  int actX = 0, actY = 0; // coordenadas actuales
  int antX = 0, antY = 0; // coordenadas anteriores
  boolean dibujo;

  public void init() {
     addMouseListener(this);
     addMouseMotionListener(this);
     dibujo = false;
  }

  /* No se usan los siguientes tres métodos, pero
     deben implementarse en null porque están
     definidos por MouseListener. */

  // Maneja entrada del ratón.
  public void mouseEntered(MouseEvent er) {
  }

  // Maneja salida del ratón.
  public void mouseExited(MouseEvent er) {
  }

  // Maneja clic del ratón.
  public void mouseClicked(MouseEvent er) {
  }

  // Maneja botón oprimido.
  public void mousePressed(MouseEvent er) {
    // guarda coordenadas
    antX = er.getX();
    antY = er.getY();
    dibujo = true;
  }

  // Maneja ratón suelto.
  public void mouseReleased(MouseEvent er) {
    dibujo = false;
  }

  // Maneja ratón arrastrado.
```

```
  public void mouseDragged(MouseEvent er) {
    // guarda coordenadas
    actX = er.getX();
    actY = er.getY();
    repaint();
  }

  // Maneja ratón movido.
  public void mouseMoved(MouseEvent er) {
    // muestra estado
    showStatus("Desplazando el ratón a " + er.getX() + ", " + er.getY());
  }

  // Despliega línea en el applet de la ventana.
  public void paint(Graphics g) {
    if(dibujo)
     g.drawLine(antX, antY, actX, actY);
  }
}
```

8. Describa brevemente la palabra **assert**.

La palabra clave **assert** crea una aseveración, que es una condición que debe ser verdadera durante la ejecución del programa. Si la aseveración es falsa, se lanza un **AssertionError**.

9. Proporcione una razón por la que un método nativo resultaría útil en algunos tipos de programas.

Un método nativo es útil cuando se hace interfaz con rutinas escritas en lenguajes diferentes de Java, o cuando se optimiza la configuración para un entorno específico en tiempo de ejecución.

Apéndice B

Uso de comentarios de documentación de Java

Como se explicó en el módulo 1, Java soporta tres tipos de comentarios. Los dos primeros son // y /* */. Al tercer tipo se le llama *comentario de documentación*, el cual empieza con la secuencia /** y termina con */. Los comentarios de documentación le permiten insertar información acerca de su programa en el programa mismo. Puede usar el programa de utilería **javadoc** (proporcionado con el JDK) para extraer la información y colocarla en un archivo HTML. Los comentarios de documentación resultan convenientes para documentar su programa. Seguramente ha visto documentación generada con **javadoc** porque es la manera en que la biblioteca de API de Java fue documentada por Sun.

Las etiquetas de javadoc

La utilería **javadoc** reconoce las siguientes etiquetas:

Etiqueta	Significado
@author	Identifica al autor de una clase.
{@code}	Despliega información tal cual, sin procesamiento de estilos de HTML, en la fuente del código. (Agregado en J2SE 5.)
@deprecated	Especifica que una clase o un miembro se ha desautorizado.
{@docRoot}	Especifica la ruta al directorio raíz de la documentación actual.
@exception	Identifica una excepción lanzada por un método.
{@inheritDoc}	Hereda un comentario de la superclase inmediata.
{@link}	Inserta un vínculo en línea con otro tema.
{@linkplain}	Inserta un vínculo en línea con otro tema, pero desplegado en una fuente de texto simple.
{@literal}	Despliega información tal cual, sin procesamiento de estilos de HTML, en la fuente del código. (Agregado en J2SE 5.)
@param	Documenta un parámetro de un método.
@return	Documenta un valor de regreso de un método.
@see	Especifica un vínculo con otro tema.
@serial	Documenta un campo predeterminado serializable.
@serialData	Documenta los datos escritos por los métodos **writeObject( )** o **writeExternal( )**.
@serialField	Documenta un componente de **ObjectStreamField**.
@since	Establece la versión cuando un cambio específico se introdujo.
@throws	Igual que **@exception**.
{@value}	Despliega el valor de una constante, que debe ser un campo **static**.
@versión	Especifica la versión de una clase.

A las etiquetas de documento que empiezan con un signo de arroba, **@**, se les denomina etiquetas *independientes* y deben usarse en su propia línea. Las etiquetas que empiezan con una llave, como **{@code}**, son etiqueta *en línea* y pueden usarse dentro de una descripción más larga. También puede usar otras etiquetas HTML estándar en un comentario de documentación. Sin embargo, no debe usar algunas etiquetas, como los encabezados, porque alteran el aspecto del archivo HTML producido por **javadoc**.

Puede usar comentarios de documentación para documentar clases, interfaces, campos, constructores y métodos. En todos los casos, el comentario de documentación debe anteceder inmediatamente al elemento que se está documentando. Cuando está documentando una variable, las etiquetas de documentación que puede usar son **@see, @since, @serial**, **@serialData**, **{@value}** y **@deprecated**. En el caso de clases, puede usar **@see**, **@author**, **@since**, **@deprecated**, **@param** y **@versión**. Los métodos pueden documentarse con **@see**, **@return**, **@param**, **@since**, **@deprecated**, **@throws**, **@serialData**, **{@inheritDoc}** y **@exception**. Una etiqueta **{@link}**, **{@docRoot}**, **{@code}**, **{@literal}** o **{@linkplain}** puede usarse en cualquier lugar. A continuación se examina cada etiqueta.

@author

La etiqueta **@author** documenta al autor de una clase. Tiene la siguiente sintaxis:

@author *descripción*

Aquí, *descripción* será usualmente el nombre de la persona que escribe la clase. La etiqueta **@author** sólo puede usarse en la documentación de una clase. Necesitará especificar la opción **-author** cuando ejecute **javadoc** para que el campo **@author** se incluya en la documentación de HTML.

{@code}

La etiqueta **{@code}** le permite incrustar texto, como un fragmento de código, en un comentario. Ese texto se despliega entonces tal cual, sin mayor procesamiento como representación en HTML, en la fuente del código. Tiene la siguiente sintaxis:

{@code *fragmento-código*}

@deprecated

La etiqueta **@deprecated** especifica que una clase o un miembro está desautorizado. Se recomienda que incluya **@see** o **{@link}** para informar al programador acerca de las opciones disponibles. La sintaxis es la siguiente:

@deprecated *descripción*

Aquí, *descripción* es el mensaje que describe la desautorización. La etiqueta **@deprecated** puede usarse en la documentación de variables, métodos y clases.

{@docRoot}

{@docRoot} especifica la ruta hacia el directorio raíz de la documentación actual.

@exception

La etiqueta **@exception** describe una excepción a un método. Tiene la siguiente sintaxis:

@exception *nombre-excepción explicación*

Aquí, el nombre plenamente calificado de la excepción se especifica en *nombre-excepción*, mientras que *explicación* es una cadena que describe la manera en que puede ocurrir la excepción. La etiqueta **@exception** sólo puede usarse en la documentación de un método.

{@inheritDoc}

Esta etiqueta hereda un comentario de una superclase inmediata.

{@link}

La etiqueta **{@link}** proporciona un vínculo en línea para información adicional. Tiene la siguiente sintaxis:

{@link *paq.clase#miembro texto*}

Aquí, *paq.clase#miembro* especifica el nombre de una clase o un método al que se le agrega un vínculo, y *texto* es la cadena que está desplegada.

{@linkplain}

La etiqueta **{@linkplain}** inserta un vínculo en línea con otro tema. El vínculo se despliega en fuente de texto simple. De otra manera, es similar a **{@link}**.

{@literal}

La etiqueta **{@literal}** le permite incrustar texto en un comentario. Luego, ese texto se despliega tal cual, sin procesamiento posterior, como representación de HTML. Tiene la siguiente sintaxis:

{@literal *descripción*}

Aquí, *descripción* es el texto que está incrustado.

@param

La etiqueta **@param** documenta un parámetro para un método. Tiene la siguiente sintaxis:

@param *nombre-parámetro explicación*

Aquí, *nombre-parámetro* especifica el nombre de un parámetro para un método, o bien, el nombre de un parámetro de tipo para una clase. El significado del parámetro está descrito por *explicación*. La etiqueta **@param** sólo puede usarse en la documentación para un método, un constructor o una clase genérica.

@return

La etiqueta **@return** describe el valor de regreso de un método. Tiene la siguiente sintaxis:

@return *explicación*

Aquí, *explicación* describe el tipo y el significado del valor regresado por un método. La etiqueta **@return** sólo puede usarse en la documentación de un método.

@see

La etiqueta **@see** proporciona una referencia a información adicional. Aquí se muestran sus formas de uso más común:

@see *ancla*

@see *paq.clase#miembro texto*

En la primera forma, *ancla* es un vínculo con un ancla o un URL relativo. En la segunda forma, *paq.clase#miembro* especifica el nombre del elemento, y *texto* es el texto desplegado para ese elemento. El parámetro de texto es opcional, y si no se usa, entonces se despliega el elemento especificado por *paq.clase#miembro*. El nombre del miembro, además, es opcional. Por lo tanto, puede especificar una referencia a un paquete, clase o interfaz además de una referencia a un método o campo específico. El nombre puede calificarse completa o parcialmente. Sin embargo, el punto que antecede al nombre del miembro (si existe) debe reemplazarse por un carácter de número (#).

@serial

La etiqueta **@serial** describe el comentario de un campo predeterminado serializable. Tiene la siguiente sintaxis:

@serial *descripción*

Aquí, *descripción* es el comentario de ese campo.

@serialData

La etiqueta **@serialData** documenta los datos escritos por los métodos **writeObject()** y **writeExternal()**. Tiene la siguiente sintaxis:

@serialData *descripción*

Aquí, *descripción* es el comentario de esos datos.

@serialField

En el caso de una clase que implementa **Serializable**, la etiqueta **@serialField** proporciona comentarios para un componente **ObjectStreamField**. Tiene la siguiente sintaxis:

@serialField *nombre tipo descripción*

Aquí, *nombre* es el nombre del campo, *tipo* es el tipo y *descripción* es el comentario de ese campo.

@since

La etiqueta **@since** establece que una clase o un miembro se introdujo en una versión específica. Tiene la siguiente sintaxis:

@since *versión*

Aquí, *versión* es una cadena que designa la versión en que se incluyó esa función. La etiqueta **@since** puede usarse en la documentación de variables, métodos y clases.

@throws

La etiqueta **@throws** tiene el mismo significado que la etiqueta **@exception**.

{@value}

{@value} tiene dos formas. La primera despliega el valor de la constante que la antecede, que debe ser un campo **static**. Tiene la forma:

{@value}

La segunda forma despliega el valor de un campo **static** especificado. Tiene esta forma:

{@value *paq.clase#campo*}

Aquí, *paq.clase#campo* especifica el nombre del campo **static**.

@version

La etiqueta **@versioón** especifica la versión de una clase. Tiene la siguiente sintaxis:

@versión *info*

Aquí, *info* es una cadena que contiene información de la versión, por lo general un número de versión, como 2.2. La etiqueta **@version** puede usarse sólo en la documentación de una clase. Usted necesitará especificar la opción **-version** cuando se ejecute **javadoc** para que el campo **@version** se incluya en la documentación de HTML.

La forma general de un comentario de documentación

Después del /** del principio, la primera o las primeras líneas se vuelven la descripción principal de su clase, variable o método. Después de eso, puede incluir una o más de las diversas etiquetas **@**. Cada etiqueta **@** debe empezar al principio de una nueva línea o ir seguida de uno o más asteriscos (*) al principio de la línea. Varias etiquetas del mismo tipo deben agruparse juntas. Por ejemplo, si tiene tres etiquetas **@see**, ponga una después de la otra. Las etiquetas en línea (las que empiezan con una llave) pueden usarse dentro de una descripción.

He aquí un ejemplo de un comentario de documentación para una clase:

```
/**
 * Esta clase dibuja una gráfica de barras
 * @author Herbert Schildt
 * @versión 3.2
*/
```

¿A qué da salida javadoc?

El programa **javadoc** toma como entrada su archivo fuente del programa de Java y da salida a varios archivos HTML que contienen la documentación del programa. La información acerca de cada clase estará en su propio archivo HTML. **javadoc** también da salida a un índice y un árbol de jerarquías. Es posible que se generen otros archivos HTML.

Un ejemplo que usa comentarios de documentación

A continuación se presenta un programa de ejemplo que usa comentarios de documentación. Observe la manera en que cada comentario antecede de inmediato el elemento que describe. Después de ser procesado por **javadoc**, la documentación acerca de la clase **NumCuadrado** se encontrará en **NumCuadrado.html**.

```
import java.io.*;
/**
 * Esta clase demuestra los comentarios de documentación.
 * @author Herbert Schildt
 * @versión 1.2
*/
public class NumCuadrado {
/**
 * Este método regresa el cuadrado de num.
 * Ésta es una descripción de varias líneas.
 * Puede usar todas las líneas que guste.
 * @param num El valor que se eleva al cuadrado.
 * @return num al cuadrado.
*/
public double square(double num) {
  return num * num;
}
/**
 * Este método da entrada a un número del usuario.
 * @return El valor de entrada como double.
 * @exception IOException En error de entrada.
 * @see IOException
*/
public double getNumber() throws IOException {
  // crea un BufferedReader empleando System.in
  InputStreamReader isr = new InputStreamReader(System.in);
  BufferedReader inData = new BufferedReader(isr);
  String cad;

  cad = inData.readLine();
  return (new Double(cad)).doubleValue();
}

/** Este método demuestra square()
/* Este método demuestra square()
```

```
 * @param args No usado.
 * @exception IOException En error de entrada
 * @see IOException
*/
public static void main(String args[])
  throws IOException
{
  SquareNum ob = new SquareNum ();
  double val;

  System.out.println("Escriba el valor que se elevará al cuadrado: ");
  val = ob.getNumber();
  val = ob.square(val);

  System.out.println("El valor al cuadrado es" + val);
  }
}
```

Índice

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X