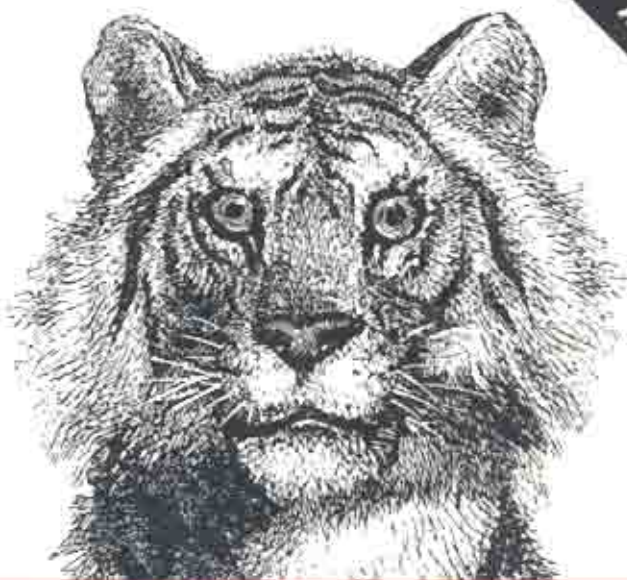


2da. edición
Actualizada



JAVA™

EN POCAS PALABRAS

Referencia al instante



O'REILLY™

David Flanagan



JAVA

EN POCAS PALABRAS

David Flanagan

Traducción:

Roberto Becerril Flores

McGRAW-HILL

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

E 28.2.04
NDP-01300

Gerente de marca: **Sergio R. Machuca Marquina**
Gerente de producto: **Fernando Castellanos Rodríguez**
Supervisor editorial: **Saúl Flores Soto**
Supervisor de producción: **Juan José García Guzmán**

621.3
FL
2
INFORMATICA
LENGUAJE DE
PROGRAMACION
TEIA
FLANAGAN

JAVA EN POCAS PALABRAS

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

DERECHOS RESERVADOS © 1999, respecto a la primera edición en español por
McGRAW-HILL INTERAMERICANA EDITORES, S.A. de C.V.

Una división de *The McGraw-Hill Companies, Inc.*

Cedro No. 512, Col. Atlampa,
Delegación Cuauhtémoc,
C.P. 06450, México, D.F.

Miembro de la Cámara Nacional de la Industria Editorial Mexicana, Reg. No. 736

ISBN 970-10-2070-7

Translated from the Second English Edition of
JAVA IN A NUTSHELL
Copyright © MCMXCVII, by O'Reilly & Associates, Inc.
All Rights reserved

ISBN 1-56592-262-X

1234567890

L.I.-98

9076543218

Impreso en México

Printed in Mexico

Esta obra se terminó de
imprimir en Septiembre de 1998 en
Litográfica Ingramex
Centeno Núm. 162-1
Col. Granjas Esmeralda
Delegación Iztapalapa
09810 México, D.F.

Se tiraron 6500 ejemplares

Contenido

<i>Prefacio</i>	<i>xi</i>
-----------------------	-----------

Parte I: Introducción a Java

<i>Capítulo 1 — Cómo comenzar con Java</i>	<i>3</i>
---	-----------------

<i>¿Por qué Java es tan interesante?</i>	<i>3</i>
--	----------

<i>Un ejemplo sencillo</i>	<i>9</i>
----------------------------------	----------

<i>Capítulo 2 — Cómo difiere Java de C</i>	<i>14</i>
---	------------------

<i>Estructura y entorno del programa</i>	<i>14</i>
--	-----------

<i>El espacio de nombre: paquetes, clases y miembros</i>	<i>16</i>
--	-----------

<i>Comentarios</i>	<i>20</i>
--------------------------	-----------

<i>Sin preprocesador</i>	<i>20</i>
--------------------------------	-----------

<i>Unicode y caracteres de escape</i>	<i>22</i>
---	-----------

<i>Tipos de datos primitivos</i>	<i>23</i>
--	-----------

<i>Tipos de datos de referencia</i>	<i>25</i>
---	-----------

<i>Objetos</i>	<i>28</i>
----------------------	-----------

<i>Arreglos</i>	<i>29</i>
-----------------------	-----------

<i>Cadenas</i>	<i>33</i>
----------------------	-----------

<i>Operadores</i>	<i>33</i>
-------------------------	-----------

<i>Declaraciones</i>	<i>36</i>
----------------------------	-----------

<i>Excepciones y manejo de excepción</i>	<i>39</i>
--	-----------

<i>Miscelánea de diferencias</i>	<i>45</i>
--	-----------

Capítulo 3 — Clases y objetos en Java 49

<i>Introducción a las clases y los objetos</i>	50
<i>Creación de un objeto</i>	52
<i>Variables de clase</i>	54
<i>Métodos de clase</i>	56
<i>Destrucción de objeto</i>	60
<i>Subclases y herencia</i>	62
<i>Invalidación de métodos</i>	68
<i>Ocultamiento y encapsulamiento de los datos</i>	71
<i>Clases e interfaces abstractas</i>	75
<i>Características de C++ no existentes en Java</i>	80
<i>Resumen</i>	81

Parte II: Introducción a Java 1.1

Capítulo 4 — Qué hay de nuevo en Java 1.1 85

<i>Java 1.1 paquete por paquete</i>	86
<i>Clases internas</i>	89
<i>El nuevo modelo de evento de AWT</i>	90
<i>Características rechazadas</i>	90
<i>Otras mejoras de AWT</i>	91
<i>Internacionalización</i>	95
<i>Serialización de objeto</i>	97
<i>Reflexión</i>	97
<i>Beans de Java</i>	98
<i>API Enterprise: JDBC, RMI y Security</i>	99
<i>Cambios de applet</i>	99
<i>Nuevas utilerías del JDK</i>	101

Capítulo 5 — Clases internas y otras características nuevas de lenguaje 102

<i>Un vistazo a las clases internas</i>	102
<i>Clases e interfaces anidadas de alto-nivel</i>	104
<i>Clases miembro</i>	106
<i>Las clases locales</i>	112
<i>Clases anónimas</i>	117
<i>Otras características nuevas de Java 1.1</i>	122

Parte III: Programación con la API de Java 1. 1

Capítulo 6 — Los applet 127

<i>Introducción a los applet</i>	<i>127</i>
<i>El primer applet</i>	<i>129</i>
<i>Cómo dibujar gráficos</i>	<i>130</i>
<i>Manejo de eventos</i>	<i>131</i>
<i>Cómo leer los parámetros del applet</i>	<i>133</i>
<i>Imágenes y sonidos</i>	<i>134</i>
<i>Archivos JAR</i>	<i>138</i>
<i>Restricciones de seguridad para los applet</i>	<i>139</i>
<i>Applet firmados</i>	<i>142</i>

Capítulo 7 — Eventos 144

<i>El modelo de evento de Java 1.0</i>	<i>144</i>
<i>Dibujando en Java 1.0</i>	<i>149</i>
<i>El modelo de evento de Java 1.1</i>	<i>150</i>
<i>Escribiendo en Java 1.1</i>	<i>153</i>
<i>Cómo escribir con las clases internas</i>	<i>154</i>
<i>Explore el modelo de evento de Java 1.1</i>	<i>155</i>

Capítulo 8 — Las nuevas características AWT..... 158

<i>El contenedor ScrollPane</i>	<i>158</i>
<i>Los menús de rápida aparición y los menús de atajo</i>	<i>159</i>
<i>Impresión</i>	<i>161</i>
<i>Cómo transferir datos con cortar-pegar</i>	<i>162</i>
<i>Demostración de las nuevas características</i>	<i>164</i>

Capítulo 9 — Serialización de objeto 172

<i>Serialización simple</i>	<i>172</i>
<i>Serialización personalizada</i>	<i>173</i>
<i>Serialización y versión de clase</i>	<i>175</i>
<i>Applet serializados</i>	<i>176</i>
<i>Serialización avanzada</i>	<i>177</i>

Capítulo 10 — Java Beans..... 178

<i>Aspectos básicos del bean</i>	179
<i>Un bean sencillo</i>	180
<i>Un bean más complejo</i>	184
<i>Eventos personalizados</i>	188
<i>Especificación de la información del bean</i>	189
<i>Definición de un editor de propiedad simple</i>	192
<i>Definición de un complejo editor de propiedad</i>	194
<i>Definición de un personalizador de bean</i>	196
<i>Convenciones y patrones de nombre</i>	199

Capítulo 11 — Internacionalización 201

<i>Unas palabras acerca de las locales</i>	201
<i>Unicode</i>	202
<i>Codificaciones de carácter</i>	207
<i>Manejo de personalización de locales</i>	209
<i>Localización de mensajes visibles al usuario</i>	211
<i>Mensajes formateados</i>	216

Capítulo 12 — Reflexión 219

<i>Cómo obtener información sobre clase y miembro</i>	219
<i>Cómo invocar un método nombrado</i>	222

Parte IV: Referencia del lenguaje Java

Capítulo 13 — Sintaxis de Java 227

<i>Tipos de datos primitivos</i>	227
<i>Secuencias de carácter de escape</i>	228
<i>Operadores</i>	228
<i>Modificadores</i>	231
<i>Palabras reservadas</i>	232
<i>Sintaxis para comentario en documentación de Java</i>	233

Capítulo 14 — Propiedades de sistema..... 235

<i>Propiedades estándares de sistema</i>	235
<i>Cómo trabajar con propiedades de sistema</i>	236

Capítulo 15 — Etiquetas de HTML relacionadas con Java 239

La etiqueta <APPLET>	239
La etiqueta <PARAM>	241
Un ejemplo de archivo HTML	241

Capítulo 16 — Herramientas del JDK 242

appletviewer - El visualizador de applet de Java	242
jar - Herramienta de archivo de Java	246
java - El intérprete de Java	247
javac - El compilador de Java	251
javadoc - El generador de documentación de Java	253
javah - Generador de archivo C por método nativo	256
javadockey - Administración de clave y firmas digitales	258
javap - El desensamblador de clase de Java	260
jdb - El depurador de Java	262
native2ascii - Convertir código fuente de Java a ASCII	266
serialver - Generador de número de versión de clase	266

Parte V: Referencia al instante de la API

Cómo usar esta referencia al instante 271

Capítulo 17 — El paquete java.applet 275

Capítulo 18 — El paquete java.awt. 279

Capítulo 19 — El paquete java.awt.datatransfer 335

Capítulo 20 — El paquete java.awt.event 340

Capítulo 21 — El paquete java.awt.image. 361

Capítulo 22 — El paquete java.awt.peer 372

Capítulo 23 — El paquete java.beans. 381

Capítulo 24 — El paquete java.io 396

Capítulo 25 — El paquete java.lang 441

Capítulo 26 — El paquete java.lang.reflect 481

Capítulo 27 — El paquete <code>java.math</code>	488
Capítulo 28 — El paquete <code>java.net</code>	492
Capítulo 29 — El paquete <code>java.text</code>	508
Capítulo 30 — El paquete <code>java.util</code>	527
Capítulo 31 — El paquete <code>java.util.zip</code>	547
Capítulo 32 — Índice de clases, métodos y campos ...	558
Índice.....	587



Prefacio

Este manual es una referencia al instante para los programadores en Java; abarca la versión 1.1 del lenguaje y API de Java. También incluye material introductorio y tutoriales para los programadores en C y C++ que quieran aprender Java. El libro está escrito para que usted lo coloque con confianza junto al teclado y lo consulte al momento de programar. El autor espera que los lectores consideren este libro útil y práctico.

Contenidos de este libro

Este libro se divide en cinco partes:

Parte I: Introducción a Java

En esta primera parte del libro está la introducción a Java y a la programación con Java, con énfasis sobre todo en ayudar a que los programadores de C y C++ puedan hacer la transición hacia Java. Si usted ya está familiarizado con la programación en Java 1.0, puede saltarse los tres capítulos de esta parte.

Parte II: Introducción a Java 1.1

Esta segunda parte del libro contiene dos capítulos que introducen las nuevas características de la API de Java 1.1, así como las nuevas características del lenguaje en Java 1.1.

Parte III: La programación con la API de Java 1.1

Esta parte contiene ejemplos de programas que muestran muchas de las nuevas características de Java 1.1. Descubrirá que esos ejemplos son un buen punto de partida para los programas que usted haga, así que adáptelos sin complicaciones para uso particular.

Parte IV: La referencia al lenguaje Java

Esta parte del libro contiene material de referencia que describe la sintaxis del lenguaje de Java, así como las herramientas que se proporcionan con el Kit de Desarrollo de Java (JDK), entre otras cosas.

Parte V: Referencia al instante para la API

Esta parte es una referencia al instante para la API de Java y constituye casi todo el libro. Por favor, lea la sección *Cómo usar esta referencia al instante*. En ella se explica cómo aprovechar al máximo el material de referencia.

Recursos sobre Java

Sun tiene documentación de referencia en línea sobre la API de Java. Dicha información puede resultar útil junto con este libro de referencia al instante. Visite <http://www.javasoft.com/> para ver o descargar esta documentación sobre la API, así como otros documentos de utilidad.

Existen muchos otros sitios en el Web que contienen información interesante sobre Java. Uno de los más conocidos es <http://www.gamelan.com/>, también conocido como <http://java.developer.com/>. Si desea discusiones (en inglés) sobre Java, visite los grupos de noticias de *com.lang.java*.*

Sitios web de Java en pocas palabras

El sitio web para este libro en inglés está en <http://www.ora.com/catalog/books/javanut2>. Ahí encontrará los ejemplos de este texto, listos para descargarse.

El sitio del autor está en <http://www.DavidFlanagan.COM/>. Es un sitio nuevo, que eventualmente contendrá un número de recursos sobre programación con Java, incluidas herramientas comerciales y de shareware, además de los “beans” que Flanagan ha escrito.

Convenciones empleadas en este libro

Las *cursivas* se usan en:

- Nombres de trayectorias, nombres de archivos y nombres de programas.
- Términos nuevos, en donde se definan.
- Direcciones de Internet, por ejemplo, nombres de dominio y URL.

Las **negritas** se usan en:

- Teclas específicas en un teclado de computadora.
- Nombres de botones y menús en interfaces de usuario.

Letter Gothic se usa en:

- Todo aquello que aparezca literalmente en un programa de Java, incluidas palabras clave, tipos de datos, constantes, nombres de métodos, variables, nombres de clase y nombres de interfaces.

- Líneas de comando y opciones que deban escribirse palabra por palabra en la pantalla.
- Todos los listados de código de Java.
- Documentos de HTML, etiquetas y atributos.

Letter Gothic Oblique se usa en:

- Parámetros de método e indicadores de que un elemento deberá reemplazarse por un valor real en el programa de usted.
- Expresiones de variable en opciones de línea de comando.

Zurich Condensado se usa en:

- Sinopsis de clase de Java en la sección de la referencia al instante. Esta fuente tan estrecha permite que usted coloque mucha información en la página, sin tener muchos cambios de línea que distraigan la atención.

Zurich Condensado bold se usa en:

- Destacado de nombres de clase, método, campo y constructor en la sección de la referencia al instante, con lo cual se vuelve más fácil examinar las sinopsis de clases.

Zurich Condensado cursivo se usa en:

- Nombres de parámetro de método y comentarios en la sección de la referencia al instante.

Reconocimientos

Mucha gente ayudó a crear este libro y por ello el autor desea darle las gracias. La deuda incluye a cientos de lectores de la primera edición en inglés, quienes escribieron para proporcionar comentarios, sugerencias, reportes de errores y elogios. Todas las contribuciones hechas están dispersas por el libro. Asimismo, el autor desea disculparse con aquellos que hicieron buenas sugerencias, pero que desafortunadamente no pudieron incluirse en esta edición.

Paula Ferguson, amiga y colega del autor, realizó las dos ediciones del libro. Las leyó cuidadosamente y siempre hizo sugerencias prácticas para hacer los libros más interesantes, claros y útiles. Ella también intervino cuando el autor comenzaba a convertir *Java en pocas palabras* en *Java en muchas palabras*.

Mike Loukides dirigió la primera edición en inglés; Eric Raymond y Troy Downing la revisaron y ayudaron a detectar errores y omisiones, además de ofrecer buenos consejos para que el libro resultase más útil a los programadores en Java.

En la segunda edición en inglés, John Zukowski revisó el material de la referencia al instante sobre el AWT de Java 1.1, mientras George Reese se encargó de revisar la parte restante del material nuevo. Esta edición también contó con el apoyo de un equipo formidable de revisores técnicos de Sun; John Rose, el autor de la “Especificación de clases internas de Java”, se encargó del capítulo que versa sobre las clases internas; Mark Reinhold, autor de “Las clases de flujo de carácter” en `java.io`, verificó la documentación del libro acerca de dichas clases; Nakul Saraiya, el diseñador de la nueva API Reflection de Java, revisó la documentación sobre el paquete `java`.

lang.reflect. El autor agradece a estos ingenieros y arquitectos, ya que los esfuerzos de todos ellos han logrado que el libro sea más preciso. Los errores que queden son atribuibles al autor.

Nicole Gipson Arigo fue la editora de producción de este libro (dio continuidad al trabajo de John Files, quien fuera editor de producción para la primera edición en inglés). Nicole coordinó todo el proceso de producción, realizó los cambios pertinentes y se encargó de corregir los cambios de renglón y de página en el manuscrito. Madeleine Newell proporcionó asistencia durante la producción del mismo. Clairemarie Fisher O'Leary, Jane Ellin y Sheryl Avruch efectuaron verificaciones de control de calidad. Seth Maislin escribió el índice. Chris Reilley creó las figuras, incluidos todos los diagramas de jerarquías de clase en la parte V.* Edie Freedman diseñó la portada. Nancy Priest diseñó el formato de los interiores del libro, al tiempo que Lenny Muellner instrumentó cuidadosamente el formato en *troff*, con la ayuda de Ellen Siever.

El autor agradece una vez más a todo el equipo de producción, pues con ayuda de todos ellos se logró conjuntar las piezas para crear el producto terminado que ahora usted tiene en las manos. Y como siempre, da las gracias y todo su amor a Christie.

David Flanagan
Abril de 1997

* Los diagramas de jerarquía se basan un poco en diagramas similares creados por Charles L. Perkins para Java 1.0.



JAVA

EN POCAS PALABRAS

PARTE I

Introducción a Java

La parte I es una introducción a Java y a la programación con Java. Si usted sabe cómo programar en C o C++, en estos capítulos aprenderá todo lo que se necesita saber para comenzar a programar con Java.

Si usted ya está familiarizado con Java 1.0, quizá quiera pasar a la parte II, que presenta las nuevas características de Java 1.1

Capítulo 1, *Cómo comenzar con Java*

Capítulo 2, *Cómo difiere Java de C*

Capítulo 3, *Clases y objetos en Java*



CAPÍTULO 1

Cómo comenzar con Java

Al darse a conocer a finales de 1995, Java tomó por asalto la Internet. Java 1.1, presentado a principios de 1997, casi duplica la velocidad del intérprete de Java e incluye muchas características nuevas e importantes. Con la adición de las API para soportar el acceso a base de datos, objetos remotos, un modelo de componente de objeto, internacionalización, impresión, criptografía, firmas digitales y muchas otras tecnologías, Java tiene como objetivo tomar por asalto el resto del mundo de la programación.

A pesar de todos los comentarios que rodean a Java y las nuevas características de Java 1.1, es importante recordar que Java, en sí, sólo es un lenguaje de programación, al igual que muchos otros, y que sus API solamente son bibliotecas de clase, como las de otros lenguajes. Lo interesante de Java, y que despierta tantos comentarios, es que tiene un número importante de características que lo hacen muy adecuado para la programación en el mundo de finales de la década de 1990, tan lleno de redes y heterogéneo. En el resto de este capítulo, se describen esas interesantes características de Java y se muestra un poco del código simple de Java. En el capítulo 4, se exploran las nuevas características agregadas a la versión 1.1 de la API de Java.

¿Por qué Java es tan interesante?

En un documento previo a la aparición de este lenguaje, Sun describió a Java de la siguiente forma:

Java: un lenguaje simple, orientado a objetos, distribuido, intérprete, robusto, seguro, de arquitectura neutra, portátil, de alto desempeño, de hilos múltiples y dinámico.

Sun reconoce que lo anterior es pura alabanza, pero el hecho es que, en su mayor parte, así se describe bien el lenguaje. Para entender por qué Java es tan interesante, ahora se examinarán las características del lenguaje que hay detrás de todas esas palabras.

Orientado a objetos

Java es un lenguaje de programación orientado a objetos. Para el programador, esto significa que debe poner atención especial a los datos de la aplicación y a los métodos que manipulan los datos, y que no debe pensar estrictamente en términos de procedimientos. Si usted está acostumbrado a la programación en C, basada en procedimiento, tal vez sienta que necesitará cambiar la manera en que usted diseña los programas al usar Java. Sin embargo, una vez que descubra cuán poderoso resulta este nuevo paradigma, pronto se acostumbrará a él.

En un sistema orientado a objetos, una *clase* es una colección de datos y métodos que operan sobre dichos datos. En conjunto, los datos y métodos describen el estado y comportamiento de un *objeto*. Las clases se ordenan en una jerarquía, para que una subclase pueda heredar el comportamiento de su superclase. Una jerarquía de clase siempre tiene una clase raíz: una clase con un comportamiento muy general.

Java posee un conjunto grande de clases, ordenadas en paquetes, que usted puede usar en los programas. Por ejemplo, Java proporciona clases que crean componentes de interfaz gráfica de usuario (el paquete `java.awt`), clases que manejan la entrada y salida (el paquete `java.io`) y clases que respaldan la funcionalidad de la red (el paquete `java.net`). La clase `Object` (en el paquete `java.lang`) sirve como la raíz de la jerarquía de clase de Java.

A diferencia de C++, Java se diseñó desde un principio para estar orientado a objetos. La mayoría de las cosas en Java son objetos; los primitivos tipos numérico, carácter y booleano son las únicas excepciones. En Java, las cadenas se representan con objetos, como sucede con otros constructores importantes de lenguaje, por ejemplo, los hilos. Una clase es la unidad básica de compilación y ejecución en Java; todos los programas de Java son clases.

Aunque Java está diseñado para verse físicamente como C++, usted descubrirá que en Java se eliminan muchas complejidades de dicho lenguaje. Si usted es un programador en C++, tal vez quiera estudiar cuidadosamente los constructores orientados a objetos en Java. Aunque la sintaxis a menudo es similar a C++, el comportamiento no es tan análogo. Si desea una descripción completa de las características de Java orientadas a objetos, consulte el capítulo 3.

Intérprete

Java es un lenguaje intérprete: el compilador de Java genera bytecode para la máquina virtual de Java (*JVM-Java Virtual Machine*), en vez de código nativo de máquina. Para que usted ejecute realmente un programa de Java, use el intérprete de Java para ejecutar los bytecode compilados. Como los bytecode de Java no dependen de plataforma, los programas de Java se pueden ejecutar en cualquier plataforma en la que la JVM se haya instalado (el intérprete y el sistema de tiempo de ejecución).

En un entorno de intérprete, se desvanece mucho la fase de “enlace” estándar de un desarrollo de programa. Si se pudiera decir que Java tiene una fase de enlace, se diría que ésta es solamente el proceso para cargar clases nuevas en el entorno; proceso de peso ligero y en incremento, que ocurre durante el tiempo de ejecución. Esto contrasta con el ciclo de compilación-enlace-ejecución de lenguajes como C y C++, que es más lento y molesto.

Arquitectura neutral y portátil

Como los programas de Java se compilan en un formato de bytecode de arquitectura neutral, una aplicación de Java se puede ejecutar en cualquier sistema, siempre y cuando dicho sistema instrumente la máquina virtual de Java. Esto resulta muy importante para las aplicaciones distribuidas en Internet u otras redes heterogéneas. Sin embargo, el enfoque de la arquitectura neutral es útil no sólo para las aplicaciones basadas en red. Si

usted es un desarrollador de aplicaciones en el mercado actual de software, probablemente quiera desarrollar versiones de aplicación que se puedan ejecutar en PC, Mac y estaciones de trabajo UNIX. Como hay distintas variedades de UNIX, Windows 95 y Windows NT en la PC, además de la nueva PowerPC de Macintosh, cada vez es más difícil producir software para todas las plataformas posibles. Sin embargo, si usted escribe una aplicación en Java, ésta se podrá ejecutar en todas las plataformas.

El hecho de que Java sea intérprete y defina un formato de bytecode estándar, de arquitectura neutral, es lo que le da la característica de ser *portátil*. Pero Java va más allá, pues asegura que no haya aspectos “dependientes de instrumentación” en la especificación del lenguaje. Por ejemplo, Java especifica explícitamente el tamaño de cada uno de los tipos de datos primitivos, así como el comportamiento aritmético de los mismos. Esto difiere de C, por ejemplo, ya que un tipo `int` puede medir 16, 32 o 64 bits de largo, dependiendo de la plataforma.

Aunque técnicamente es posible escribir programas no portátiles en Java, resulta muy sencillo evitar las pocas dependencias de plataforma que expone la API de Java y escribir así programas de Java “puros” o verdaderamente portátiles. El programa nuevo de Sun, “100% Java Puro” (“100% *Pure Java*”), ayuda a que los desarrolladores aseguren (y certifiquen) que su código sea portátil. Para ser coherentes con el lema de Sun, “Una vez escrito, se ejecuta en todas partes”, los programadores sólo necesitan hacer esfuerzos sencillos para evitar errores que no permitan al programa ser portátil.

Dinámico y distribuido

Java es un lenguaje *dinámico*. En todo momento, una clase de Java se puede cargar en un intérprete de Java en ejecución. Dinámicamente se pueden crear casos de estas clases cargadas. Las bibliotecas de código nativo también se pueden cargar de manera dinámica. Las clases en Java se representan mediante la clase `Class`; usted puede obtener información dinámicamente acerca de una clase al momento de la ejecución, en especial en Java 1.1, con la adición de la API Reflection (Reflexión), descrita en el capítulo 12.

A Java también se le denomina lenguaje *distribuido*. Esto significa, simplemente, que proporciona un soporte de alto nivel para redes. Por ejemplo, la clase `URL` y las clases relacionadas en el paquete `java.net` hacen que la lectura de un archivo o una fuente remota sea tan fácil como leer un archivo local. De manera similar, en Java 1.1, la API para Invocación de Método Remoto (*Remote Method Invocation*, RMI) permite que un programa de Java llame a métodos de objetos Java remotos, como si se tratara de objetos locales. (Java también proporciona soporte de red tradicional, de bajo nivel, incluidos datagramas y conexiones basadas en flujo por socket.)

La naturaleza distribuida de Java realmente se aprecia cuando se combina con sus capacidades de cargado dinámico de clase. En conjunto, estas características hacen posible que un intérprete de Java descargue y ejecute código desde Internet. (Como se verá más adelante, Java instrumenta severas medidas de seguridad para verificar que esto se haga sin peligro.) Esto sucede cuando un navegador del Web descarga y ejecuta un applet de Java, por ejemplo. Sin embargo, los escenarios pueden ser más complejos. Imagine un procesador de multimedia escrito en Java. Cuando se pida a este programa que despliegue algún tipo de datos que nunca antes haya enfrentado, el programa podrá descargar dinámicamente una clase desde la red, que pudiera analizar los datos, para después descargar dinámicamente otra clase (quizá un Java “bean”) que despliegue los datos dentro de un documento compuesto. Un programa como éste usa recursos distribuidos en la red, para crecer y adaptarse dinámicamente a las necesidades del usuario.

Simple

Java es un lenguaje *simple*. Los diseñadores de Java intentaron crear un lenguaje que un programador pudiera aprender con rapidez, por lo que el número de constructores del lenguaje se ha mantenido relativamente bajo. Otra meta del diseño fue lograr que el

lenguaje fuese familiar a la mayoría de los programadores, para facilitar la migración. Si usted es un programador en C o C++, descubrirá que Java utiliza muchos de los mismos constructores de lenguaje que C y C++.

Para que el lenguaje fuese pequeño y familiar, los diseñadores de Java eliminaron varias características disponibles en C y C++. Estas características, en su mayoría, conducen a prácticas pobres de programación o se usan rara vez. Por ejemplo, Java no soporta la declaración `goto`; en vez de eso, proporciona declaraciones con etiqueta `break` y `continue`, así como manejo de excepciones. Java tampoco usa archivos de encabezado y elimina el preprocesador C. Como Java está orientado a objetos, los constructores de C como `struct` y `union` se han eliminado. Java también elimina las características de sobrecargado y herencia múltiple de C++.

Sin embargo, quizá la simplificación más importante es que Java no usa punteros. Los punteros son de los aspectos más propensos a errores en la programación en C y C++. Como Java no tiene estructuras, y los arreglos y las cadenas son objetos, no hay necesidad de punteros. Java maneja automáticamente por usted las referencias y no referencias de los objetos. Java también efectúa recolección automática de basura, para que usted no se preocupe por asuntos de la administración de la memoria. Todo esto lo libera a usted de tener que preocuparse por punteros pendientes, referencias a punteros no válidos y fugas de memoria, para que invierta su tiempo en desarrollar la funcionalidad de los programas.

Suena como si Java hubiera dejado a C y C++ detrás, y como si solamente hubiera un shell de lenguaje de programación. Sin embargo, no hay que adelantar juicios. Como se verá en el capítulo 2, Java en realidad es un lenguaje con características completas y muy elegante.

Robusto

Java se ha diseñado para escribir software robusto y muy confiable. Java ciertamente no elimina la necesidad de asegurar la calidad del software, ya que todavía es posible escribir software defectuoso en Java. Sin embargo, Java sí elimina ciertos tipos de errores de programación, cosa que hace considerablemente más sencilla la escritura de software confiable.

Java es un lenguaje de tipografía rígida, por lo que permite una verificación exhaustiva al tiempo de la compilación, en busca de posibles problemas de no concordancia de letras. La tipografía de Java es más severa que la de C++, el cual hereda de C un número de laxitudes de tiempo de compilación, especialmente en el área de las declaraciones de función. Java requiere declaraciones explícitas de método; no soporta declaraciones implícitas al estilo de C. Estos requerimientos son más estrictos y aseguran que el compilador atrape errores en la invocación del método, con lo que se consiguen programas más confiables.

Una de las cosas que hace simple a Java es la carencia de punteros y aritmética de puntero. Esta característica también incrementa la robustez de los programas de Java, al abolir una clase entera de defectos relacionados con los punteros. De manera similar, todos los accesos a arreglos y cadenas se verifican al tiempo de la ejecución, para asegurar que estén en límites; así se elimina la posibilidad de sobrescribir memoria y corromper los datos. Las formas de los objetos de un tipo a otro también se verifican al momento de la ejecución, para asegurar que sean legales. Por último —y esto es muy importante—, la recolección automática de basura en Java evita que la memoria tenga fugas, así como la presencia de otros defectos perniciosos relacionados con la asignación y desasignación de memoria.

El manejo de excepción es otra característica de Java que contribuye a formar programas más robustos. Una *excepción* es una señal de que ha ocurrido una condición excepcional, por ejemplo, un error de “no se puede encontrar el archivo”. Al usar la declaración `try/catch/finally`, usted puede agrupar todo el código de manejo de

error en un solo lugar, con lo que se simplifica mucho la tarea del manejo y recuperación de errores.

Seguro

Uno de los aspectos más resonados de Java es que se trata de un lenguaje *seguro*. Esto es especialmente importante, dada la naturaleza distribuida de Java. Sin un aseguramiento de la seguridad, usted no querría descargar código de un sitio al azar en la Internet, ni lo dejaría ejecutar en su computadora. Sin embargo, esto es lo que la gente hace con los applet de Java todos los días. Java se diseñó con la seguridad en mente, y proporciona varias capas de controles de seguridad que protegen contra código malicioso; estas capas permiten a los usuarios ejecutar con comodidad programas desconocidos, como los applet.

En el nivel más bajo, la seguridad va de la mano con la robustez. Como ya se ha visto, los programas de Java no pueden falsificar punteros en la memoria ni sobresaturar los arreglos o leer memoria fuera de los límites de un arreglo o cadena. Estas características son de las defensas principales de Java contra el código malicioso. Al no permitir por completo ningún acceso directo a la memoria, se desecha una clase enorme y confusa de ataques a la seguridad.

La segunda línea de defensa contra el código malicioso es el proceso de verificación bytecode, que el intérprete de Java ejecuta en todo código no probado que se cargue. Estos pasos de verificación aseguran que el código esté bien formado; es decir, que no sature o desprovea a la pila, y que no contenga bytecode ilegales, por ejemplo. Si el paso de verificación de bytecode se salta, cualquier bytecode elaborado maliciosamente o corrupto puede aprovechar las debilidades de instrumentación en un intérprete de Java.

Otra capa de protección de seguridad se conoce comúnmente como el "modelo del cajón de arena": todo código indigno de confianza se coloca en un "cajón de arena", donde se ejecuta con toda seguridad, sin dañar al "mundo real" ni al entorno completo de Java. Cuando un applet, u otro código no probado, se ejecuta en el cajón de arena, hay restricciones en cuanto a lo que éste puede hacer. La más obvia de estas restricciones es que no tiene acceso alguno al sistema de archivo local. Existen otras restricciones en el cajón de arena, que las lleva a cabo una clase *SecurityManager*. El modelo funciona, ya que todas las clases centrales de Java que efectúan operaciones sensibles, como el acceso al sistema de archivo, primero piden permiso al *SecurityManager* instalado en ese momento. Si un código no digno de confianza realiza la llamada, de manera directa o indirecta, el administrador de seguridad lanza una excepción y no se permite la operación. Vea el capítulo 6 si desea una lista completa de las restricciones puestas a los applet que se ejecutan en el cajón de arena.

Finalmente, en Java 1.1 hay otra posible solución al problema de la seguridad. Al anexar una firma digital al código de Java, el origen de dicho código se puede establecer en una manera criptográficamente segura e imborrable. Si usted ha especificado que confía en una persona u organización, entonces el código que lleva la firma digital de dicha entidad confiable se vuelve digno de confianza, aun cuando se cargue a través de la red, y quizá se pueda ejecutar sin las restricciones del modelo del cajón de arena.

Sin embargo, claro, la seguridad no es una cosa definitiva. Así como no se puede garantizar que un programa esté 100% libre de defectos, ningún lenguaje o entorno es 100% seguro. Pero a pesar de lo dicho, Java, al parecer, sí ofrece un nivel práctico de seguridad para la mayoría de las aplicaciones. Se anticipa y defiende contra casi todas las técnicas usadas históricamente para engañar al software y hacer que se comporte mal, además de haber sido escrutinado intensamente por expertos de seguridad y *hackers*. En las versiones tempranas de Java se descubrieron algunos agujeros de seguridad, pero dichas fallas se repararon casi de inmediato. Parece razonable esperar que los posibles agujeros que aparezcan en el futuro, también se corrijan muy rápido.

Alto desempeño

Java es un lenguaje intérprete, así que nunca será tan rápido como un lenguaje compilador, por ejemplo, C. Se decía que Java 1.0 era cerca de 20 veces más lento que C. Sin embargo, Java 1.1 es casi el doble de rápido que Java 1.0, así que se puede decir que el código de C compilado se ejecuta diez veces más rápido que los bytecode interpretados de Java. Pero, antes de que usted levante los brazos en señal de disgusto, sepa que esta velocidad es más que adecuada para ejecutar aplicaciones interactivas, basadas en red y GUI, donde la aplicación a menudo permanece desocupada, a la espera de que el usuario haga algo, o en espera de datos de la red. Por otra parte, las secciones críticas de velocidad del entorno de tiempo de ejecución de Java, que hacen cosas como la concatenación y comparación de cadena, se instrumentan con código nativo eficiente.

Conforme el desempeño se mejora, muchos intérpretes de Java incluyen ahora compiladores "justo a tiempo" que pueden traducir los bytecode de Java en código de máquina para un CPU, en particular al tiempo de la ejecución. El formato bytecode de Java fue diseñado con estos compiladores "justo a tiempo" en mente, así que el proceso para generar código de máquina es bastante eficaz y produce código razonablemente bueno. De hecho, Sun declara que el desempeño de los bytecode convertidos a código máquina es tan bueno como C o C++. Si usted está dispuesto a sacrificar la portabilidad del código para ganar velocidad, también podrá escribir porciones de un programa en C o C++ y usar métodos nativos de Java para hacer interfaz con este código nativo.

Cuando se toma en cuenta el desempeño, es importante recordar qué aspectos de Java son débiles en el espectro de los lenguajes de programación disponibles. En un extremo del espectro, hay lenguajes de argumento de alto nivel, completamente intérpretes, por ejemplo, Tcl y los shell de UNIX. Estos lenguajes son grandiosos para hacer prototipos y son muy portátiles, pero también muy lentos. En el otro extremo del espectro, están los lenguajes compiladores de bajo nivel, como C y C++. Estos lenguajes ofrecen alto desempeño, pero no son muy portátiles ni buenos en términos de confiabilidad. Java se sitúa en la mitad del espectro. El desempeño de los bytecode interpretados de Java es mucho mejor que el de los lenguajes de argumento de alto nivel (incluso Perl), además de ofrecer la sencillez de estos lenguajes y de ser portátil.

Hilos múltiples

En una aplicación de red basada en GUI, como lo es un navegador del Web, es fácil imaginar que varias cosas se lleven a cabo a un mismo tiempo. Un usuario podría escuchar un clip de audio mientras se desplaza por una hoja, y en el plano secundario el navegador descargaría una imagen. Java es un lenguaje de *hilos múltiples*; proporciona soporte para varios hilos de ejecución (en ocasiones llamados procesos de peso ligero) que pueden manejar diferentes tareas. Un beneficio importante de los hilos múltiples es que se mejora el desempeño interactivo de las aplicaciones gráficas para el usuario.

Si usted ha intentado trabajar con hilos en C o C++, sabe que esto puede ser muy difícil. Java hace que la programación con hilos sea mucho más sencilla, pues proporciona soporte de lenguaje incorporado para los hilos. El paquete `java.lang` ofrece una clase `Thread` que soporta métodos para comenzar y detener hilos, y para establecer propiedades de hilo, entre otras cosas. La sintaxis del lenguaje Java soporta hilos directamente con la palabra clave `synchronized`. Esta palabra clave vuelve en extremo sencillo marcar secciones de código o métodos enteros que sólo se deban ejecutar por un hilo a la vez.

Aunque los hilos son cosas "de magia" en C y C++, en Java son de uso común. Dado que Java hace tan sencillo emplear los hilos, las bibliotecas de clase de Java requieren su uso en un cierto número de lugares. Por ejemplo, todo applet que ejecute animación lo logra mediante un hilo. De manera similar, Java no soporta E/S sin bloqueo, asíncrona,

con notificación a través de señales o interrupciones; se debe crear en su lugar un hilo que se bloquee en cada canal de E/S que a usted le interese.

Un ejemplo sencillo

Ahora, usted seguramente ya tiene una buena idea de por qué Java es un lenguaje tan interesante. Por ello no se hablará más de conceptos abstractos y sí se examinará algo de código concreto de Java. Sin embargo, antes de mirar un applet interesante, se pagará tributo al applet favorito y ubicuo denominado "Hola mundo".

Hola mundo

El ejemplo 1-1 muestra el programa de Java más simple que existe: "Hola mundo".

Ejemplo 1-1: Hola mundo

```
public class HolaMundo {  
    public static void main(String[] args) {  
        System.out.println("¡Hola Mundo!");  
    }  
}
```

Este programa, al igual que cualquier programa de Java, consiste en una definición de clase pública. La clase contiene un método denominado `main()`, que es el punto de entrada principal para todas las aplicaciones de Java; es decir, el punto desde el cual el intérprete comienza a ejecutar el programa. El cuerpo de `main()` consiste en una línea, que imprime la salida del mensaje:

```
¡Hola Mundo!
```

Este programa se debe guardar en un archivo con el mismo nombre que la clase pública, más una extensión `.java`. Para compilarlo, usted usaría `javac`.*

```
% javac HolaMundo.java
```

Este comando produce el archivo `HolaMundo.class` en el directorio actual. Para ejecutar el programa, use usted el intérprete de Java, `java`:

```
% java HolaMundo
```

Observe que, cuando usted invoca al intérprete, no se proporciona la extensión `.class` al archivo que quiere ejecutar.

Un applet para escribir

El ejemplo 1-2 muestra un programa de Java menos trivial. Este programa es un applet y no sólo una aplicación individual como el programa "Hola mundo" que se vio con anterioridad. Dado que este ejemplo es un applet, tiene una estructura diferente de la de la aplicación individual; en concreto, tiene un método `main()`. Al igual que todos los applet, éste se ejecuta dentro de un visualizador de applet o navegador del Web, y permite que el usuario dibuje (o escriba) con el ratón, como se muestra en la figura 1-1.

* Siempre y cuando usted use el Java Development Kit (JDK, Kit de desarrollo de Java) de Sun. Si usted usa un entorno de desarrollo de Java proveniente de otro vendedor, siga las instrucciones de dicho vendedor.

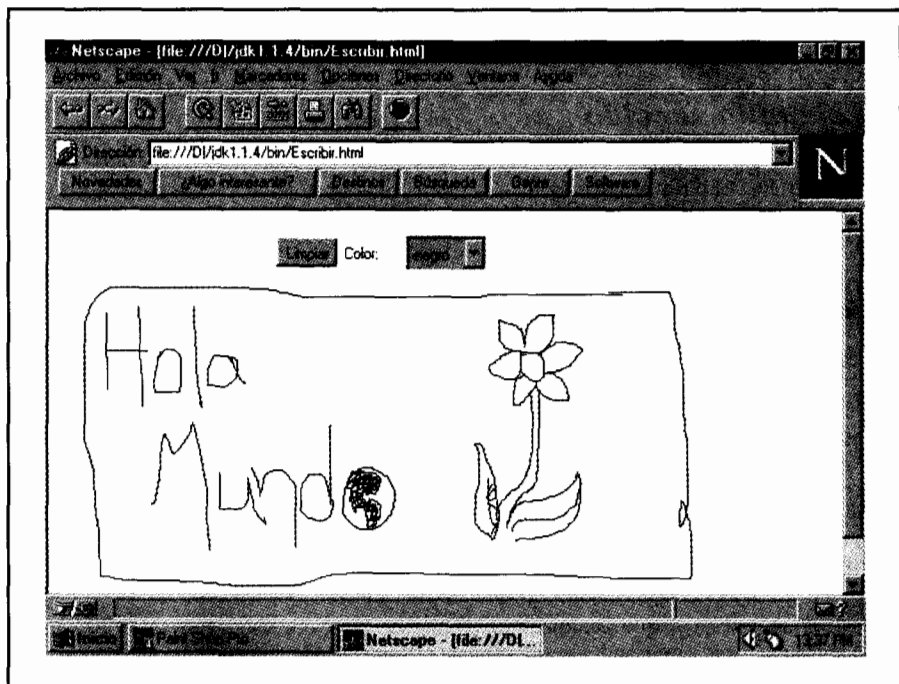


Figura 1-1: Un applet de Java en ejecución dentro de un navegador del Web.

Uno de los cambios más importantes entre Java 1.0 y Java 1.1 es la manera en que se notifican los “eventos” a los programas de Java, por ejemplo, un movimiento del ratón. En el ejemplo 1-2 se usa el modelo de evento de Java 1.0, en vez del modelo de evento preferido de Java 1.1. Esto se debe a que la generación actual de navegadores del Web (al momento de escribirse este libro) todavía usa Java 1.0. Para lograr que este applet se use ampliamente, está codificado con el modelo de evento “menospreciado”.*

Ejemplo 1-2: Un applet de Java

```
import java.applet.*;
import java.awt.*;

public class Escribir extends Applet {
    private int ultimo_x, ultimo_y;           //Almacena la ultima
                                              // posicion del raton.
    private Color color_actual = Color.black; // Almacena el color actual.
```

*Si a usted le interesa actualizar este programa para usar Java 1.1, en el capítulo 7 busque información sobre cómo usar el nuevo modelo de evento de 1.1. Por otra parte, usted necesita cambiar la llamada de `bounds()` en el método `action()`, por `getBounds()`, en caso de querer evitar una advertencia de compilación por usar un método menospreciado.

Ejemplo 1-2: un applet de Java (continuación)

```
private Button boton_limpiar;           // El boton Limpiar.
private Choice opciones_color;          // La lista descendente
                                         // de color.

// Este metodo es llamado para inicializar el applet.
// Los applet no tienen un metodo main().
public void init() {
    // Pone el color de fondo
    this.setBackground(Color.white);

    // Crea un boton y lo agrega al applet. Establece los colores del boton
    boton_limpiar = new Button("Limpiar");
    boton_limpiar.setForeground(Color.black);
    boton_limpiar.setBackground(Color.lightGray);
    this.add(boton_limpiar);

    // Crea un menu de colores y lo agrega al applet.
    // Tambien pone los colores del menu y agrega una etiqueta.
    opciones_color = new Choice();
    opciones_color.addItem("negro");
    opciones_color.addItem("rojo");
    opciones_color.addItem("amarillo");
    opciones_color.addItem("verde");
    opciones_color.setForeground(Color.black);
    opciones_color.setBackground(Color.lightGray);
    this.add(new Label("Color: "));
    this.add(opciones_color);
}

// Este metodo es llamado cuando el usuario hace clic con el raton para
// empezar a escribir.
public boolean mouseDown(Event e, int x, int y)
{
    ultimo_x = x; ultimo_y = y;
    return true;
}

// Este metodo es llamado cuando el usuario arrastra el raton.
public boolean mouseDrag(Event e, int x, int y)
{
    Graphics g = this.getGraphics();
    g.setColor(color_actual);
    g.drawLine(ultimo_x, ultimo_y, x, y);
    ultimo_x = x;
    ultimo_y = y;
    return true;
}

// Este metodo es llamado cuando el usuario hace clic sobre el boton o
// escoge un color
public boolean action(Event event, Object arg) {
    // Si el boton Limpiar fue presionado, se maneja aqui.
    if (event.target == boton_limpiar) {
        Graphics g = this.getGraphics();
        Rectangle r = this.bounds();
        g.setColor(this.getBackground());
    }
}
```

```

        g.fillRect(r.x, r.y, r.width, r.height);
        return true;
    }
    // De otra forma si el color fue seleccionado, se maneja
    else if (event.target == opciones_color) {
        if (arg.equals("negro")) color_actual = Color.black;
        else if (arg.equals("rojo")) color_actual = Color.red;
        else if (arg.equals("amarillo")) color_actual = Color.yellow;
        else if (arg.equals("verde")) color_actual = Color.green;
        return true;
    }
    // De otra forma, se deja a la superclase manejarlo.
    else return super.action(event, arg);
}
}

```

No espere comprender todo el applet en este punto. Sólo servirá como antecedente del lenguaje. En el capítulo 2 y en el capítulo 3 se explicarán los constructores de lenguaje que usted necesita para entender el ejemplo. Posteriormente, en el capítulo 6 y en el capítulo 7 se explicarán los conceptos de applet y el manejo de evento empleados en este ejemplo.

Lo primero que usted debe notar al analizar el código es que se parece mucho a C y C++. Las declaraciones `if` y `return` resultan familiares. En la asignación de valores a las variables se utiliza la sintaxis esperada. Los procedimientos (llamados “métodos” en Java) son reconocibles como tales.

El segundo aspecto que habrá de notarse es la naturaleza orientada a objetos del código. Como se puede ver en la parte inicial del ejemplo, el programa consiste en la definición de una clase pública. El nombre de la clase que se está definiendo es `Escribir`; es una extensión, o subclase, de la clase `Applet`. (El nombre completo de la clase `Applet` es `java.applet.Applet`. Una de las declaraciones `import`, al principio del ejemplo, permite referencia a `Applet` con su abreviatura.)

Se dice que las clases “encapsulan” los datos y métodos. Como se puede ver, la clase `Escribir` contiene declaraciones de variable y de método. Los métodos en realidad se definen dentro de la clase. Los métodos de una clase a menudo se invocan mediante un ejemplo de la clase. Por ello, se ven líneas como:

```
opciones_color.addItem("negro");
```

Esta línea de código invoca al método `addItem()` del objeto al que se hace referencia, mediante la variable `opcion_color`. Si usted es un programador en C, pero no un programador en C++, quizá le cueste trabajo acostumbrarse a esta sintaxis. Se verán más ejemplos de ello en los capítulos 2 y 3. Observe que `this` es una palabra clave, no un nombre de variable; se refiere al objeto actual y en este ejemplo, al objeto `Escribir`.

El método `init()` de un applet es llamado por el navegador del Web o visualizador de applet, al comenzar el applet. En el ejemplo, este método crea un botón **Borrar** y un menú de opciones de color, para después agregar estos componentes de GUI al applet.

Los métodos `mouseDown()` y `mouseDrag()` se llaman cuando el usuario hace clic y arrastra el ratón. Éstos son los métodos encargados de dibujar líneas conforme el usuario escribe con el ratón. El método `action()` se llama cuando el usuario hace clic sobre el botón **Borrar** o selecciona un color en el menú de colores. El cuerpo del método determina cuál de estos dos “eventos” ha ocurrido y maneja el evento, según corresponda. Recuerde que estos métodos son parte del modelo de evento de Java 1.0. En el capítulo 7 se explica este modelo y también el modelo de evento de Java 1.1, que lo reemplaza.

Para compilar este ejemplo, usted lo guardaría en un archivo llamado *Escribir.java* y usaría *javac*:

```
% javac Escribir.java
```

Este ejemplo es un applet, no un programa individual como el ejemplo "Hola Mundo". No tiene un método *main()* y, por lo tanto, no lo puede ejecutar directamente el intérprete de Java. En vez de eso, se debe hacer referencia a él en un archivo de HTML y ejecutar el applet en un visualizador de applet o navegador del Web. El visualizador de applet o el navegador del Web carga la clase en el intérprete de Java que se esté ejecutando, e invoca los distintos métodos del applet en los momentos adecuados. Para incluir el applet en una página del Web, se usaría un fragmento de HTML como el siguiente:

```
<APPLET code="Escribir.class" width=500 height=300>
</APPLET>
```

El ejemplo 1-3 muestra un archivo completo de HTML que se podría usar para desplegar el applet. En el capítulo 15 se explica la sintaxis de HTML para los applet, con todo detalle.

Ejemplo 1-3: un archivo de HTML que contiene un applet

```
<HTML>
<HEAD>
<TITLE> El Applet Escribir </TITLE>
</HEAD>
<BODY>
Por favor escribe en el applet de abajo.
<P>
<APPLET code="Escribir.class" width=500 height=300>
¡Lo sentimos! Tu navegador no soporta Java, o Java no esta
habilitado.
</APPLET>
</BODY>
</HTML>
```

Suponga que usted guarda este archivo de HTML como *Escribir.html*. Posteriormente, para ejecutar el applet, usted podría usar el comando de Sun *appletviewer*, así:

```
% appletviewer Escribir.html
```

También podría desplegar el applet al visualizar el archivo *Escribir.html* en el navegador del Web, siempre y cuando el navegador respalde los applet de Java. En la figura 1-1 se mostró el applet Escribir en ejecución en Netscape Navigator.



CAPÍTULO 2

Cómo difiere Java de C

Java es muy parecido a C, por lo que resulta relativamente sencillo que los programadores de C lo aprendan. Sin embargo, existen diferencias importantes entre C y Java, como la carencia de un preprocesador, el uso de caracteres Unicode de 16 bits y el mecanismo de manejo de excepción. ¡En este capítulo se explican dichas diferencias, para que los programadores que ya conozcan C, puedan comenzar a programar en Java de inmediato!

En este capítulo también se señalan las similitudes y diferencias entre Java y C++. Sin embargo, los programadores en C++ deben tener cuidado: aunque Java toma prestada mucha terminología e incluso sintaxis de C++, las analogías entre Java y C++ no son por ningún motivo tan marcadas como las existentes entre Java y C. Los programadores en C++ no deben dejarse deslumbrar por un falso sentimiento de familiaridad con Java, sólo porque los lenguajes compartan un cierto número de palabras clave.

Una de las áreas principales en las que Java difiere de C, por supuesto, es que Java es un lenguaje orientado a objetos y tiene mecanismos para definir clases y crear objetos que sean instancias de dichas clases. Las características orientadas a objetos de Java son tema para un capítulo entero, por lo que se explicarán con detalle en el capítulo 3.

Estructura y entorno del programa

Un programa en Java consiste en una o más definiciones de clase, cada una de las cuales ha sido compilada en su propio archivo *.class* de código objeto de la máquina virtual de Java. Una de estas clases debe definir un método *main()*, donde el programa comienza a ejecutarse.*

Para llamar un programa de Java, usted debe ejecutar el intérprete de Java, *java*, y especificar el nombre de la clase que contenga el método *main()*. Deberá omitir la extensión *.class* al hacer esto. Observe que un applet de Java no es una aplicación, sino una clase de Java que se carga y ejecuta por una aplicación de Java que ya se encuentra en ejecución, por ejemplo, un navegador del Web o un visualizador de applet.

* *Método* es un término orientado a objetos para un procedimiento o una función. Usted lo verá empleado en todo este libro.

El método `main()`, que el intérprete de Java invoca para comenzar un programa de Java, deberá tener el prototipo siguiente:

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

El intérprete de Java se ejecuta una vez que se regresa el método `main()`, o cuando el intérprete llega al final de `main()`. Si el programa no ha creado hilos, el intérprete termina el programa. De no ser así, el intérprete continúa en ejecución hasta que se termina el último hilo.

Argumentos de la línea de comando

El argumento simple para `main()` es un arreglo de cadenas, convencionalmente denominadas `args` o `argv`. La longitud de este arreglo (que se pasaría como el argumento `argc` en C) está disponible como `argv.length`, como ocurre con cualquier arreglo de Java. Los elementos del arreglo son los argumentos, de haberlos, que aparecieron en la línea de comando del intérprete después del nombre de la clase. Observe que el primer elemento del arreglo *no* es el nombre de la clase, como lo esperaría un programador en C. El ejemplo 2-1 muestra cómo podría usted escribir un comando *echo* al estilo de UNIX (un programa que simplemente imprime sus argumentos) en Java.

Ejemplo 2-1: Un programa de echo en Java

```
public class echo {  
    public static void main (String argv[]){  
        for (int i=0; i < argv.length; i++)  
            System.out.print(argv[i] + " ");  
        System.out.print("\n");  
        System.exit(0);  
    }  
}
```

Valor de salida del programa

Observe que `main()` se debe declarar para regresar `void`. Por ello, usted no puede regresar un valor desde el programa de Java con una declaración `return` en `main()`. Si necesita regresar un valor, llame a `System.exit()` con el valor entero deseado, como se ha hecho en el ejemplo 2-1. Note que el manejo y la interpretación de este valor de salida dependen, por supuesto, del sistema operativo. `System.exit()` provoca que el intérprete de Java termine el programa de inmediato, sin importar que estén o no otros hilos en ejecución.

Entorno

La API de Java no permite que un programa de Java lea variables del entorno del sistema operativo, ya que dependen de la plataforma. Sin embargo, Java define un mecanismo similar, independiente de la plataforma, conocido como *la lista de propiedades de sistema*, para asociar valores de texto con nombres.

Un programa de Java puede buscar el valor de una propiedad nombrada con el método `System.getProperty()`:

```
string dircasa = System.getProperty("usuario.casa");  
string debug = System.getProperty("miapp.debug");
```

Al iniciar, el intérprete de Java automáticamente define un número de propiedades estándar de sistema. Al especificar la opción `-D` al intérprete, usted puede insertar definiciones de propiedad adicionales a la lista:

```
%java -Dmiapp.debug=true miapp
```

Consulte el capítulo 14 si desea más información sobre las propiedades del sistema.

El espacio de nombre: paquetes, clases y miembros

Como lenguaje diseñado para soportar cargado dinámico de módulos por toda la Internet, Java tiene especial cuidado en evitar conflictos de espacio de nombre. Las variables globales simplemente no son parte del lenguaje. Tampoco lo son las funciones “globales” ni los procedimientos.

Sin variables globales

En Java, cada campo y método se declaran dentro de una clase y forman parte de dicha clase. Asimismo, cada clase es parte de un *paquete* (en Java 1.1, las clases también se pueden declarar dentro de otras clases). Los campos y métodos (y las clases en 1.1) de una clase se conocen como los *miembros* de una clase. A todo campo o método de Java se puede hacer referencia mediante su nombre totalmente calificado, que consiste en el nombre del paquete, el nombre de la clase y el nombre del miembro (esto es, el nombre del campo o método), separados por puntos. Los nombres de paquete se forman generalmente de componentes múltiples, separados por puntos. Así, el nombre totalmente calificado de un método podría ser:

```
david.juegos.tetris.EfectosDeSonido.tocar()
```

Nombres de archivo y estructura de directorio en Java

Un archivo de código fuente de Java tiene la extensión *.java*. Consiste en una declaración `package` opcional, con cierto número de declaraciones `import`, seguidas por una o más definiciones de clase o interfaz. (Las declaraciones `package` e `import` se presentarán más adelante.) Si en un archivo fuente de Java está definida más de una clase o interfaz, sólo una de ellas se puede declarar `public` (esto es, sólo una está disponible fuera del paquete); además, el archivo fuente debe tener el mismo nombre que la clase o interfaz pública, y la extensión *.java*.

Cada definición de clase o interfaz en un archivo *.java* es compilada en un archivo separado. Estos archivos de bytecode de Java compilados se conocen como “archivos de clase” y deben tener el mismo nombre que la clase o interfaz que definan, con la extensión *.class* anexada. Por ejemplo, la clase `EfectosDeSonido` se almacenaría en el archivo *EfectosDeSonido.class*.

Los archivos de clase se almacenan en un directorio que tiene los mismos componentes que el nombre del paquete. Si el nombre totalmente calificado de una clase es `david.juegos.tetris.EfectosDeSonido`, por ejemplo, la trayectoria completa del archivo de clase debe ser *david/juegos/tetris/EfectosDeSonido.class*.* Este nombre de archivo se interpreta en relación con la “trayectoria de clase” de Java, que se describe más adelante.

* En este libro se usarán especificaciones de directorio al estilo UNIX. Si usted es un programador en Windows, simplemente cambie todas las diagonales normales de los archivos por diagonales invertidas. De manera similar, en las especificaciones de trayectoria, cambie los signos de dos puntos por punto y coma.

Paquetes de la API de Java

La API de Java 1.1 consiste en las clases e interfaces definidas en los veintitrés paquetes listados en la tabla 2-1.

Tabla 2-1: Los paquetes de la API de Java

Nombre del paquete	Contenidos
java.applet	Clases de applet
java.awt	Clases de gráficos, ventana y GUI
java.awt.datatransfer	Clases de transferencia de datos (por ejemplo, cortar y pegar)
java.awt.event	Clases e interfaces para procesamiento de evento
java.awt.image	Clases para procesamiento de imagen
java.awt.peer	Interfaces GUI para independencia de plataforma
java.beans	API del modelo de componente JavaBeans
java.io	Tipos diversos de clases de entrada y salida
java.lang	Clases nucleares de lenguaje
java.lang.reflect	Clases API de reflejo
java.math	Aritmética de precisión arbitraria
java.net	Clases de red
java.rmi	Clases para Invocación de Método Remoto
java.rmi.dgc	Clases relacionadas con RMI
java.rmi.registry	Clases relacionadas con RMI
java.rmi.server	Clases relacionadas con RMI
java.security	Clases de seguridad
java.security.acl	Clases relacionadas con seguridad
java.security.interfaces	Clases relacionadas con seguridad
java.sql	API SQL JDBC para acceso a base de datos
java.text	Clases de internacionalización
java.util	Distintos tipos de datos útiles
java.util.zip	Clases para compresión y descompresión

La trayectoria de clase de Java

El intérprete de Java sabe dónde están instaladas sus clases de sistema estándares, y las carga desde dicho sitio, conforme las necesite. Por omisión, busca las clases definidas por el usuario que estén en el directorio actual o cerca de éste. Usted puede establecer la variable de entorno CLASSPATH para decir al intérprete dónde buscar las clases definidas por el usuario. El intérprete siempre anexa la ubicación de sus clases de sistema, al final de la trayectoria especificada por esta variable de entorno. Las entradas en una especificación de trayectoria de clase deben ser directorios o archivos ZIP que contengan las clases. Los directorios en una especificación de trayectoria de clase deben estar separados por signos de dos puntos en un sistema UNIX, y separados por punto y coma en un sistema de Windows. Por ejemplo, en un sistema UNIX, usted podría usar:

```
setenv CLASSPATH ./home/david/classes:/usr/local/javatools/classes.zip
```

En un sistema Windows podría usarse:

```
set CLASSPATH .;C:\david\classes;D:\local\javatools\classes.zip
```

Esto indica a Java que debe buscar clases que no sean de sistema en y debajo de los directorios especificados. Observe que el directorio actual (.) está incluido en estas trayectorias.

Usted también puede especificar una trayectoria de clase al intérprete de Java con el argumento de línea de comando `-classpath`. Al especificar esta opción se invalida toda trayectoria especificada y la variable de entorno `CLASSPATH`. Note que el intérprete no anexa la ubicación de las clases de sistema al final de esta trayectoria, por lo que usted mismo debe especificar dichas clases de sistema. Finalmente, observe que el compilador de Java también reconoce y hace honores a las trayectorias de clase especificadas con la variable de entorno `CLASSPATH` y el argumento de línea de comando `-classpath`.

Nombres de paquete globalmente únicos

Los diseñadores de Java han propuesto un esquema de nomenclatura de paquete único, válido en toda la Internet y basado en el nombre de dominio de Internet de la organización donde se desarrolla el paquete.

La figura 2-1 muestra algunos nombres totalmente calificados, que incluyen componentes de paquete, clase y campo.

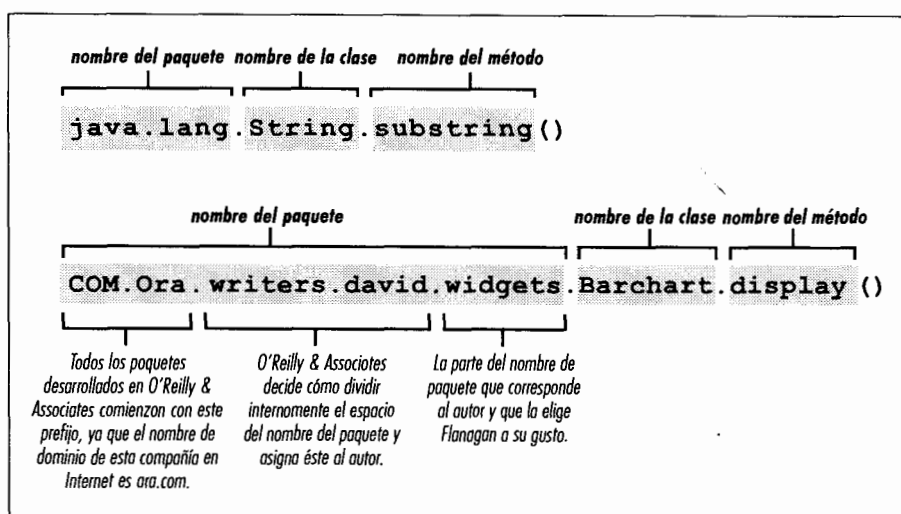


Figura 2-1: Nombres totalmente calificados en Java

Algunas organizaciones ya siguen este esquema de nomenclatura y producen clases con nombres como `com.sybase.jdbc.SybDriver`. Sin embargo, otra tendencia en desarrollo consiste en que las compañías simplemente usen el nombre de la empresa como primer componente de los nombres de paquete, y generen clases como `netscape.javascript.JSObject`.

Y claro, los nombres de paquete de nivel superior `java` y `sun` están reservados para uso exclusivo de Sun. Los desarrolladores no deben definir nuevas clases dentro de estos paquetes.

La declaración `package` (paquete)

La declaración `package`, de haberla, debe aparecer como la primera declaración (esto es, el primer texto, aparte de los comentarios y el espacio en blanco) en un archivo de código fuente de Java. Sirve para especificar de qué paquete es parte el código en el archivo. El código de Java que sea parte de un paquete en particular tiene acceso a

todas las clases (`public` y no `public`) del paquete, y a todos los métodos y campos no `private` en dichas clases. Cuando el código de Java es parte de un paquete nombrado, el archivo de clase compilado se debe colocar en la posición adecuada en la jerarquía de directorio `CLASSPATH`, antes de que el intérprete de Java u otras utilerías puedan tener acceso a él.

Si la declaración `package` se omite en un archivo, el código en dicho archivo es parte de un paquete no nombrado por omisión. Este hecho resulta conveniente en programas pequeños de prueba o durante el desarrollo, ya que significa que el código se puede interpretar desde el directorio actual.

La declaración import (importar)

La declaración `import` hace que las clases de Java estén disponibles para la clase actual, bajo un nombre abreviado. Las clases públicas de Java siempre están disponibles vía sus nombres totalmente calificados, siempre y cuando el archivo de la clase en cuestión se pueda encontrar (y sea legible), respecto a la variable de entorno `CLASSPATH`. `import` en realidad no hace que la clase esté disponible o “leíble”; tan sólo evita que usted teclee y hace que el código sea más legible.

En un programa de Java puede aparecer cualquier número de declaraciones `import`. Sin embargo, deben aparecer después de la declaración opcional `package`, al comienzo del archivo y antes de la primera definición de clase o de interfaz en el archivo.

Hay dos formas de la declaración `import`:

```
import package.class;  
import package.*;
```

La primera forma permite que la clase especificada en el paquete especificado se conozca por su nombre de clase simplemente. Así, esta declaración `import` permite que usted escriba `Hashtable` en vez de `java.util.Hashtable`:

```
import java.util.Hashtable;
```

La segunda forma de la declaración `import` logra que todas las clases de un paquete estén disponibles mediante su nombre de clase. Por ejemplo, la declaración `import` siguiente está implícita (usted no necesita especificarla por sí mismo) en todo programa de Java:

```
import java.lang.*;
```

De esta manera, las clases centrales del lenguaje están disponibles por sus nombres de clase sin calificar. Si dos paquetes importados con esta forma de declaración contienen clases con el mismo nombre, se cometerá un error al usar cualquiera de esas clases ambiguas, sin emplear su nombre totalmente calificado.

Acceso a paquetes, clases y miembros de clase

Java tiene las siguientes reglas referentes al acceso a paquetes, clases y miembros de clase. (Los miembros de clase son las variables, los métodos y, en Java 1.1, las clases anidadas definidas dentro de una clase.) Las palabras clave `public`, `private` y `protected` empleadas en estas reglas se explicarán con más detalle en el capítulo siguiente.

- Un paquete está disponible si los archivos y directorios en cuestión están disponibles (por ejemplo, si los archivos locales tienen permisos de lectura adecuados o se pueden descargar vía red).
- Todas las clases e interfaces de un paquete están disponibles para todas las otras clases e interfaces del mismo paquete. No es posible definir clases en Java, visibles solamente dentro de un solo archivo de código fuente.

- Una clase declarada **public** en un paquete está disponible dentro de otro paquete, siempre y cuando el paquete en sí también esté disponible. Una clase no **public** no está disponible desde fuera de su paquete.
- Los miembros de una clase están disponibles desde una clase diferente dentro del mismo paquete, siempre y cuando no estén declarados como **private**. Los miembros **private** sólo están disponibles dentro de su propia clase.
- Un miembro de una clase **A** está disponible desde una clase **B** en un paquete diferente, si **A** es **public** y el miembro es **public**, o si **A** es **public**, el miembro es **protected** y **B** es una subclase de **A**.
- Todos los miembros de una clase siempre están disponibles desde dicha clase.

Variables locales

Las reglas para nombre de espacio que se han descrito se aplican a paquetes, a clases y a los miembros de las clases. Java también soporta variables locales, declaradas dentro de definiciones de método. Estas variables locales se comportan como las variables locales en C; es decir, ni tienen nombres jerárquicos globalmente únicos ni modificadores de acceso como **public** y **private**. Las variables locales son totalmente diferentes de los campos de clase.

Comentarios

Java soporta tres tipos de comentarios:

- Un comentario estándar al estilo de C que comience con **/*** y continúe hasta el siguiente ***/**. Como en la mayoría de las instrumentaciones de C, este estilo de comentario no se puede anidar.
- Un comentario al estilo de C++ que comience con **//** y continúe hasta el final de la línea.
- Un “comentario de documentación” especial que comience con **/**** y continúe hasta el siguiente ***/**. Estos comentarios no se anidan. Los comentarios de documentación los procesa especialmente el programa *javadoc*, para producir simple documentación en línea del código fuente de Java. Consulte el capítulo 13 si desea saber más sobre la sintaxis del comentario de documentación, y el capítulo 16 si necesita información sobre el programa *javadoc*.

Como los comentarios al estilo de C no se anidan, es conveniente usar comentarios **//** al estilo de C++ para la mayoría de los comentarios cortos que se hacen dentro de los cuerpos de método. Esto le permite a usted emplear comentarios **/**** para comentar grandes bloques de código cuando sea necesario, durante el desarrollo. Esto es muy importante, ya que, como se verá, Java no soporta un preprocesador que permita el uso de **#if 0** para comentar un bloque.

Sin preprocesador

Java no incluye ningún tipo de preprocesador, como el preprocesador *cpp* de C. Parece difícil imaginar la programación sin **#define**, **#include** y **#ifdef**; pero, de hecho, Java no requiere estos constructores.

Definición de constantes

En Java, toda variable declarada `final` es una constante; es decir, su valor se debe especificar con un inicializador cuando sea declarada, y dicho valor nunca podrá cambiar. El equivalente en Java de una constante definida con `#define` de C es una variable `static final`, declarada dentro de una definición de clase. Si el compilador puede computar el valor de dicha variable `static final` al tiempo de la compilación, utiliza el valor computado para precomputar otras constantes durante la compilación, referentes al valor. La variable `java.lang.Math.PI` es un ejemplo de dicha constante. Se declara de esta manera:

```
public final class Math{
    ...
    public static final double PI=3.14159.....;
    ...
}
```

Observe dos cosas en este ejemplo. Primero, la convención de C sobre usar letras MAYÚSCULAS para las constantes también es una convención de Java. Segundo, las constantes de Java tienen una ventaja sobre otras constantes de preprocesador de C: las constantes de Java tienen nombres jerárquicos globalmente únicos, mientras las constantes definidas con el preprocesador de C siempre corren el riesgo de una colisión de nombres. Por otra parte, las constantes de Java tienen una tipografía severa y permiten que el compilador verifique mejor la ortografía, en comparación con las constantes del preprocesador de C.

Definición de macros

El preprocesador de C permite que usted defina macros; es decir, que defina un constructor que parezca una invocación de función, pero que en realidad sea reemplazada directamente por código de C, para evitar la sobrecarga de una llamada de función. Java no tiene equivalente en este tipo de macro, pero la tecnología de compilador ha avanzado hasta el punto en que las macros rara vez son necesarias. Un buen compilador de Java debe ser capaz, automáticamente, de "poner en línea" métodos pequeños de Java donde sea apropiado.

Inclusión de archivos

Java no tiene una directiva `#include`, pero no la necesita. Java define un mapeo de nombres de clases totalmente calificados (como `java.lang.Math`) para una estructura de directorio de archivo (como `java/lang/Math.class`). Esto significa que cuando el compilador de Java debe leer un archivo de clase especificada, sabe exactamente dónde encontrarlo y no requiere una directiva especial que le diga dónde buscar.

Por otra parte, Java no distingue la diferencia entre *declarar* una variable o procedimiento y *definirla(o)*, como sucede en C. Esto significa que no hay necesidad de usar archivos de encabezado al estilo de C o prototipos de función; es decir, un solo archivo de objeto de Java sirve como definición de interfaz e instrumentación de una clase.

Java sí tiene una declaración `import`, superficialmente similar a la directiva `#include` de preprocesador de C; sin embargo, esta declaración dice al compilador si el archivo actual está usando las clases especificadas o las clases del paquete especificado, y permite que usted se refiera a dichas clases con nombres abreviados. Por ejemplo, como el compilador importa (`imports`) implícitamente todas las clases del paquete `java.lang`, usted puede hacer referencia a la constante `java.lang.Math.PI` mediante el nombre más corto `Math.PI`.

Compilación condicional

Java no tiene ninguna forma de las directivas `#ifdef` o `#if` de C para efectuar compilación condicional. En teoría, la compilación condicional no es necesaria en Java porque, al ser un lenguaje independiente de la plataforma, no hay dependencias de plataforma que requieran la técnica. Sin embargo, en la práctica, la compilación condicional todavía es útil en Java para proporcionar interfaces de usuario ligeramente diferentes en las distintas plataformas, por ejemplo, o para proporcionar la inclusión opcional de código corrector de errores en los programas.

Aunque Java no define constructores explícitos para compilación condicional, un buen compilador de Java (por ejemplo, *javac* de Sun) efectúa implícitamente compilación condicional; es decir, no compila código si puede probar que el código jamás será ejecutado. En general, esto significa que no se incluye el código dentro de una declaración `if` que pruebe una expresión que siempre resulte `false`. Así, la colocación de código dentro de un bloque `if (false)` equivale a rodearlo con `#if 0` y `#endif` en C.

La compilación condicional también funciona con las constantes que, como ya se vio, son variables `static final`. Una clase puede definir la constante así:

```
private static final boolean DEBUG = false;
```

Con dicha constante definida, cualquier código que esté en un bloque `if (DEBUG)` realmente no se compila en el archivo de clase. Para activar la eliminación de errores de la clase, sólo es necesario cambiar el valor de la constante por `true` y recompilar la clase.

Unicode y caracteres de escape

Los caracteres, las cadenas y los identificadores de Java (por ejemplo, nombres de variable, método y clase) se componen de caracteres Unicode de 16 bits. Esto hace que los programas de Java sean relativamente sencillos de internacionalizar para los usuarios que no hablan inglés; también hace que el lenguaje sea más amigable con los programadores que tampoco hablan inglés. Así, un programador de Tailandia podría usar el alfabeto tailandés para los nombres de clase y método de su código de Java.

Si los caracteres de dos bytes le parecen confusos o atemorizantes, no tenga miedo. El conjunto de carácter de Unicode es compatible con ASCII y los primeros 256 caracteres (0x0000 a 0x00FF) son idénticos a los caracteres ISO8859-1 (Latín-1), del 0x00 a 0xFF. Además, el diseño del lenguaje Java y la API `String` de Java hacen que la representación de carácter sea completamente transparente para usted. Si usted sólo usa caracteres Latín-1, no hay forma de que distinga incluso entre un carácter de Java de 16 bits y los caracteres de 8 bits a los que está acostumbrado. Para más información sobre el Unicode, consulte el capítulo 11.

La mayoría de las plataformas no puede desplegar todos los 38,885 caracteres de Unicode definidos actualmente, por lo que los programas de Java podrían escribirse (y la salida de Java aparecería) con secuencias de escape especiales de Unicode. En cualquier parte de un programa de Java (no sólo dentro de las literales de carácter y cadena), un carácter de Unicode se puede representar con la secuencia de escape de Unicode `\uxxxx`, donde `xxxx` corresponde a una secuencia de cuatro dígitos hexadecimales.

Java también soporta todas las secuencias de escape de carácter estándar de C, por ejemplo, `\n`, `\t`, y `\xxx` (donde `\xxx` son tres dígitos octales). Note, sin embargo, que Java no soporta la continuación de línea con `\` al final de una línea. Las cadenas largas deben ser especificadas como una sola línea larga, o crearse a partir de cadenas cortas con el operador de concatenación de cadena (+). (Observe que dos cadenas de constante se concatenan al momento de la compilación y no al tiempo de la ejecución, por lo que usar el operador + de esta manera generará buenos resultados.)

Existen dos diferencias importantes entre los escapes de Unicode y los caracteres de escape al estilo de C. La primera, como se ha dicho, consiste en que los escapes de Unicode pueden aparecer en cualquier parte de un programa de Java, mientras los otros caracteres de escape pueden aparecer solamente en constantes de carácter y cadena.

La segunda diferencia, más sutil, es que las secuencias de escape `\u` de Unicode se procesan antes que los otros caracteres de escape, por lo que los dos tipos de secuencias de escape pueden tener semánticas muy diferentes. Un escape de Unicode es simplemente una manera alternativa de representar un carácter que quizá no se desplegaría en ciertos sistemas (que no sean de Unicode). Sin embargo, algunos escapes de carácter representan caracteres especiales, de una forma que evita la interpretación normal de dichos caracteres por parte del compilador. Los ejemplos siguientes sirven para explicar esta diferencia. Observe que `\u0022` y `\u005c` son los escapes de Unicode para el carácter de comillas dobles y el carácter de diagonal invertida, respectivamente.

```
// \" representa un caracter ", y previene la interpretacion
// normal de dicho caracter por el compilador.
// Esta es una cadena que consiste en un caracter de dobles comillas.
String comilla = "\"";

// No podemos representar la misma cadena con un simple escape Unicode.
// \u0022 tiene el mismo significado para el compilador que ".
// La cadena de abajo se convierte en "": una cadena vacía seguida
// por una cadena no terminada, que genera un error de compilación.
String quote = "\u0022";

// Aquí representamos ambos caracteres de un escape \" como
// escapes Unicode. Se convierte en "\"", y es la misma
// cadena como en el primer ejemplo.
String quote = "\u005c\u0022";
```

Tipos de datos primitivos

Java agrega los tipos primitivos `byte` y `boolean` al conjunto estándar de tipos de C. Asimismo, define estrictamente el tamaño y signo de sus tipos. En C, un `int` puede ser de 16, 32 o 64 bits, y un `char` puede actuar con o sin signo, dependiendo de la plataforma. Pero eso no sucede en Java. En C, una variable local no inicializada tiene por lo general basura como valor. En Java, todas las variables tienen valores garantizados por omisión, aunque el compilador pueda avisarle a usted en qué lugares usted mismo depende, por accidente o no, de estos valores por omisión. La tabla 2-2 lista los tipos de datos primitivos de Java. Las subsecciones posteriores proporcionan detalles acerca de estos tipos.

Tabla 2-2: Tipos de datos primitivos de Java

Tipo	Contiene	Omisión	Tamaño	Valor mín	Valor máx
booleano	true o false	false	1 bit	N.A.	N.A.
char	Carácter de unicode	\u0000	16 bits	\u0000	\uFFFF
byte	entero con signo	0	8 bits	-128	127

Tipo	Contiene	Omisión	Tamaño	Valor mín	Valor máx
short	entero con signo	0	16 bits	-32768	32767
int	entero con signo	0	32 bits	-2147483648	2147483647
long	entero con signo	0	64 bits	-9223372036854775808	9223372036854775807
float	IEEE 754 punto flotante	0.0	32 bits	$\pm 1.40239846E-45$	$\pm 3.40282347E+38$
double	IEEE 754 punto flotante	0.0	64 bits	$\pm 4.94065645841246544E-324$	$\pm 1.79769313486231570E+308$

El tipo booleano

Los valores `boolean` no son enteros, no pueden tratarse como enteros y nunca serán calculados para o a partir de cualquier otro tipo. Para efectuar conversiones al estilo de C entre un valor `boolean` y uno `int i`, se usa el siguiente código:

```
b=(i !=0); //entero-a-booleano: distinto-0 ->true; 0 ->false;
i=(b)?1:0; //booleano-a-entero: true ->1; false ->0;
```

El tipo char

Los valores `char` representan caracteres. En un programa de Java, las literales de carácter pueden aparecer entre comillas sencillas. Por ejemplo:

```
char c= 'A';
```

Todos los escapes de carácter estándar de C, así como los escapes de Unicode, también se respaldan en literales de carácter. Por ejemplo:

```
char nuevalinea = '\n', apostrofo = '\'', borrar = '\377', aleph = '\u05D0';
```

Los valores del tipo `char` no tienen signo. Si un valor `char` se calcula en un `byte` o un `short`, puede resultar un valor negativo.

En Java, el tipo `char` guarda un carácter Unicode de dos bytes. Aunque esto pareciera extraño para quienes no están familiarizados con el Unicode y las técnicas de internacionalización de programa, es totalmente transparente. Java no proporciona una manera para computar el tamaño de una variable, ni permite algún tipo de aritmética de puntero. Esto quiere decir que si usted sólo está usando caracteres ASCII o Latín-1, no hay forma de distinguir un `char` de Java de un `char` de C.

Tipos enteros

Los tipos enteros en Java son `byte`, `short`, `char`, `int` y `long`. Las literales para estos tipos se escriben justo como en C. Todos los tipos enteros, excepto `char`, tienen signo. No hay palabra clave `unsigned` (sin signo) como sucede en C. No es válido escribir `long`

`int` o `short int` como lo es en C. Una constante `long` se puede distinguir de otras constantes enteras al anexarle el carácter `l` o `L` a la misma.

La división de entero entre cero o entre módulo cero ocasiona que se lance una `ArithmeticException` (Excepción aritmética).*

Tipos de punto flotante

Los tipos de punto flotante en Java son `float` y `double`. Las literales para estos tipos se escriben como en C y se pueden especificar para que sean del tipo `float`, al anexar una `f` o `F` al valor; también se pueden especificar para que sean del tipo `double` al anexar una `d` o `D`.

Los tipos `float` y `double` tienen valores especiales que a veces son el resultado de ciertas operaciones de punto flotante: infinidad positiva, infinidad negativa, cero negativo y no número. Las clases `java.lang.Float` y `java.lang.Double` definen algunos de estos valores como constantes: `POSITIVE_INFINITY`, `NEGATIVE_INFINITY`, y `NaN`.

`NaN` no está ordenada y en comparación con cualquier otro número, incluida ella misma, genera `false`. Use `Float.isNaN()` o `Double.isNaN()` para probar `NaN`. El cero negativo compara a igualdad con el cero regular (cero positivo), pero los dos ceros se pueden distinguir entre sí por división: uno dividido por cero negativo genera infinidad negativa; uno dividido entre cero positivo genera infinidad positiva.

La aritmética de punto flotante nunca causa excepciones, incluso en el caso de la división entre cero.

Literales de cadena

Las cadenas en Java no son un tipo primitivo, pero son instancias de la clase `String`. Sin embargo, debido a que se usan mucho, las literales de cadena pueden aparecer entre comillas en los programas de Java, como sucede en C. Cuando el compilador encuentra una literal de cadena, automáticamente crea el objeto `String` necesario.

Tipos de datos de referencia

Los tipos de datos no primitivos en Java son los objetos y arreglos. Estos objetos no primitivos a menudo se denominan “tipos de referencia”, ya que se manejan “por referencia”. En otras palabras, la dirección del objeto o arreglo está almacenada en una variable, que se pasa a los métodos, y así sucesivamente. Por comparación, los tipos primitivos se manejan “por valor”; es decir, los valores primitivos actuales están almacenados en variables y se pasan a los métodos.

En C, usted puede manipular un valor por referencia al tomar la dirección de éste con el operador `&`, y puede “desreferenciar” una dirección con los operadores `*` y `->`. Estos operadores no existen en Java: los tipos primitivos siempre se pasan por valor; los arreglos y objetos siempre se pasan por referencia.

Como los objetos se pasan por referencia, dos variables diferentes pueden referirse al mismo objeto:

```
Button p, q;  
p = new Button();           // p hace referencia al objeto Button  
q = p;                       // q hace referencia al mismo Button.
```

*Las excepciones señalan errores en Java. El manejo de excepción se describe posteriormente en este capítulo.

```
p.setLabel("Ok");           // Un cambio al objeto a traves de p...
String s = q.getLabel();     // ...tambien se ve a traves de q.
                             // s contiene ahora "Ok".
```

Sin embargo, no sucede lo mismo con los tipos primitivos:

```
int i = 3;                   // i contiene el valor de 3.
int j = i;                   // j contiene una copia del valor en i.
i = 2;                       // el cambio en i no cambia a j.
                             // ahora, i == 2 y j == 3.
```

Terminología: pasar por referencia

La aseveración de que Java manipula objetos “por referencia” confunde a algunos programadores, ya que existen varios significados para la frase “por referencia” en el habla común. Sin importar el término, es importante entender lo que hace Java: funcionar con referencias a objetos. Una variable de Java almacena solamente una referencia a un objeto y no al objeto en sí. Cuando se pasa un objeto a un método, sólo se pasa en realidad la referencia al objeto, no todo el objeto. En este sentido, Java manipula objetos “por referencia”.

Algunas personas usan el término “pasar por referencia” para explicar que una referencia a una variable se pasa a un método. Java no hace esto. Por ejemplo, en Java *no* es posible escribir una función de trabajo `swap()` como la siguiente:

```
public void swap(Object a, Object b){
    Object temp = a;
    a = b;
    b = temp;
}
```

Los parámetros de método `a` y `b` contienen referencias a objetos, no direcciones de variables. Así, mientras esta función `swap()` se compila y ejecuta, no tiene efecto sino sólo en sus propias variables y argumentos locales.

Para poner fin a este problema de terminología, quizá debiera decirse que Java manipula objetos “por referencia”, pero pasa “por valor” dichas referencias a los métodos.

Copiado de objetos

Como los tipos de referencia no se pasan por valor, al asignar un objeto a otro en Java no se copia el valor de dicho objeto; simplemente se asigna una referencia al objeto. Examine el siguiente código:

```
Button a = new Button("Ok");
Button b = new Button("Cancelar");
a = b;
```

Después de que se ejecutan estas líneas, la variable `a` contiene una referencia al objeto a que `b` hace referencia. El objeto al que `a` solía hacer referencia se ha perdido.

Para copiar los datos de un objeto a otro, use el método `clone()`:

```
Vector b = new Vector();
c = b.clone();
```

Después de ejecutarse estas líneas, la variable `c` hace referencia a un objeto que es el duplicado del objeto a que `b` hace referencia. Observe que no todos los tipos soportan el método `clone()`; solamente las clases que instrumentan la interfaz `Cloneable` se

pueden duplicar. Para más información sobre la duplicación de objetos, lea sobre `java.lang.Cloneable` y `java.lang.Object.clone()` en el capítulo 25.

Los arreglos también son tipos de referencia, y al asignar un arreglo nada más copian una referencia al arreglo. Para copiar realmente los valores almacenados en un arreglo, usted debe asignar cada uno de los valores de manera individual o usar el método `System.arraycopy()`.

Cómo verificar la igualdad de los objetos

Otra implicación de pasar objetos por referencia es que el operador `==` prueba si dos variables hacen referencia al mismo objeto, no si dos objetos contienen los mismos valores. Para probar si dos objetos separados son en realidad los mismos, use un método especialmente escrito para dicho tipo de objeto (justo como usaría `strcmp()` para comparar la igualdad de cadenas de C). En Java, un número de clases define un método `equals()`, factible de usar para efectuar esta prueba.

Java no tiene punteros

En Java, la referencia y desreferencia de objetos se maneja automáticamente para usted. Java no permite que el usuario manipule punteros o direcciones de memoria de ningún tipo:

- No acepta que usted lance referencias de objeto o arreglo a enteros o viceversa.
- No permite que usted haga aritmética de puntero.
- No deja que usted compute el tamaño en bytes de ningún tipo primitivo u objeto.

Existen dos razones para estas restricciones:

- Los punteros constituyen una fuente notoria de errores. Al eliminarlos, se simplifica el lenguaje y se acaban muchos errores potenciales.
- Los punteros y la aritmética de puntero se podrían usar para evitar que ocurran verificaciones al tiempo de la ejecución y que funcionen los mecanismos de seguridad en Java. Eliminar punteros permite que Java proporcione las garantías de seguridad ofrecidas.

Para un programador en C, la ausencia de punteros y de la aritmética de puntero pudiera parecer una restricción odiosa en Java. Sin embargo, cuando usted se acostumbre al modelo de programación orientada a objetos de Java, ésta ya no le parecerá en absoluto una restricción seria. La carencia de punteros significa que usted probablemente no podrá hacer ciertas cosas, como escribir manejadores de dispositivo UNIX en Java (por lo menos, no sin usar métodos nativos escritos en C). No obstante, así está bien, ya que la mayoría de los programadores nunca tiene por qué hacer este tipo de programación de bajo nivel.

nulo

El valor por omisión de las variables de todos los tipos de referencia es `null`. `null` es un valor reservado que indica “una ausencia de referencia”; es decir, que una variable no hace referencia a ningún objeto o arreglo.

En Java, `null` es una palabra clave reservada, a diferencia de `NULL` en C, donde sólo es una constante definida para ser 0. `null` es una excepción a las reglas estrictas de tipografía de Java; se puede asignar a cualquier variable de tipo de referencia (esto es, cualquier variable que tenga una clase, una interfaz o un arreglo como su tipo).

`null` no se puede lanzar a cualquier tipo primitivo, incluidos los tipos enteros y booleanos. No se debe considerar como igual a cero (aunque se puede instrumentar de esa manera).

Resumen del tipo de referencia

La distinción entre tipos primitivos pasados por valor y los objetos y arreglos pasados por referencia, es de gran importancia en Java. Asegúrese de entender lo siguiente:

- En Java, todos los objetos y arreglos se manejan por referencia (sin embargo, dichas referencias a objeto se pasan por valor a los métodos).
- Los operadores `=` y `==` asignan y prueban las referencias a objetos. Use `clone()` y `equals()` para copiar o probar realmente los objetos en sí.
- La referencia y la desreferencia necesarias para los objetos y arreglos las maneja Java automáticamente.
- Un tipo de referencia nunca se puede lanzar a un tipo primitivo.
- Un tipo primitivo nunca se puede lanzar a un tipo de referencia.
- No hay aritmética de puntero en Java.
- No hay operador `sizeof` en Java.
- `null` es un valor especial que significa "sin objeto" o que indica la ausencia de referencia.

Objetos

Ahora que usted sabe que los objetos se pasan por referencia, se describirá la manera en que éstos se crean, usan y destruyen. En las subsecciones siguientes se proporciona un resumen muy breve de los objetos; pero en el capítulo 3 se explican las clases y los objetos con mucho más detalle.

Creación de objetos

Al declarar una variable para que guarde un objeto no se crea al objeto en sí; la variable sólo aloja la referencia al objeto. Para crear realmente un objeto, usted debe usar la palabra clave `new`. Ésta va seguida de la clase del objeto (esto es, su tipo) y una lista de argumento opcional en paréntesis. Estos argumentos se pasan al método de constructor para la clase, y sirven para inicializar campos internos en el objeto nuevo. Por ejemplo:

```
java.awt.Button b = new java.awt.Button();  
ComplexNumber c = new ComplexNumber(1.0, 1.414);
```

Existen en realidad dos maneras extra para crear un objeto. Primera, usted puede crear un objeto `String` con sólo encerrar los caracteres en comillas dobles:

```
String s="Esto es un texto"
```

Como las cadenas se usan con mucha frecuencia, el compilador de Java proporciona esta técnica como un atajo. Otra manera de crear objetos consiste en llamar al método `newInstance()` de un objeto `Class`. En general, esta técnica sólo se usa cuando se cargan clases dinámicamente, por lo cual no se describirá aquí. En Java 1.1, los objetos también se pueden crear al "deserializarlos"; esto es, al recrear un objeto cuyo estado se haya guardado mediante "serialización".

La memoria para los objetos recién creados se asigna de manera dinámica. Crear un objeto con **new** en Java es como llamar **malloc()** en C, para asignar memoria a una instancia de un **struct**. Y claro, también se acerca mucho al uso del operador **new** en C++. (Más adelante, se verá en qué punto desaparece esta analogía en **malloc()** en C y **new** en C++.)

Acceso a objetos

Como usted probablemente ya se dio cuenta, en los distintos fragmentos de código vistos hasta ahora, los campos de un objeto se accesan con un punto:

```
ComplexNumber c = new ComplexNumber();  
c.x = 1.0;  
c.y = -1.414;
```

Esta sintaxis es la reminiscencia del acceso a los campos de una **struct** en C. Sin embargo, recuerde que los objetos de Java siempre se accesan por referencia y que Java efectúa toda la desreferenciación necesaria por usted. Así, el punto en Java es más como **->** en C. Java oculta el hecho de que hay una referencia aquí, en un intento por hacer más sencilla la programación para usted. La otra diferencia entre C y Java, al accederse un objeto, es que en Java usted hace referencia a los métodos de un objeto con la misma sintaxis empleada para los campos:

```
ComplexNumber c = new ComplexNumber(1.0,-1.414);  
double magnitud = c.magnitud();
```

Recolección de basura

Los objetos en Java se crean con la palabra clave **new**, pero no hay una palabra clave **old** o **delete** ni un método **free()** para deshacerse de ellos cuando ya no sean necesarios. Si crear un objeto con **new** es como llamar a **malloc()** en C o usar **new** en C++, entonces parecería que Java está lleno de fugas de memoria, ya que nunca se llama a **free()** ni se usa el operador **delete**.

Sin embargo, éste no es el caso. Java usa una técnica denominada *garbage collection* (recolección de basura) para detectar y liberar automáticamente los objetos en desuso (un objeto ya no está en uso cuando no hay más referencias al mismo). Esto significa que en los programas que usted acaba de adquirir, nunca tendrá que preocuparse por liberar memoria o destruir objetos, pues el recolector de basura se encargará de ello.

Si usted es un programador en C o C++, tal vez necesite tiempo para acostumbrarse a que los objetos asignados se vayan, sin reclamar la memoria que ocuparon. Entonces, cuando se acostumbre, comenzará a apreciar la belleza de esta característica. En el capítulo siguiente se describirá la recolección de basura con más detalle.

Arreglos

La mayoría de lo que usted aprendió en las secciones previas acerca de los tipos de referencia y objetos, también se aplica a los arreglos en Java:

- Los arreglos se manipulan por referencia.
- Se crean dinámicamente con **new**.
- Se recolectan automáticamente como basura, cuando ya no se hace referencia a ellos.

En las subsecciones siguientes se explican éstos y otros detalles.

Creación y destrucción de arreglos

Existen dos maneras de crear arreglos en Java. En la primera se usa `new` y se especifica cuán largo debe ser el arreglo:

```
byte buffer_octal[] = new byte[1024];
Button botones[] = new Button[10];
```

Como al crear un arreglo no se crean los objetos almacenados en el arreglo, no hay que llamar a un constructor y la lista de argumento se omite con esta forma de la palabra clave `new`. Los elementos de un arreglo creado de esta manera se inicializan en el valor por omisión, de acuerdo con su tipo. Los elementos de un arreglo de `int` se inicializan en 0, por ejemplo, y los elementos de un arreglo de objetos se inicializan en `null`. Este último punto no debe olvidarse: al crearse un arreglo de objetos solamente se asigna almacenamiento para las referencias de objeto, no para los objetos en sí. Los objetos a que se hará referencia mediante los elementos del arreglo se deben crear por separado.

La segunda manera como se crea un arreglo es mediante un inicializador, similar al de C:

```
int tabla_busqueda[]={1, 2, 4, 8, 16, 32, 64, 128};
```

Esta sintaxis crea dinámicamente un arreglo e inicializa sus elementos en los valores especificados. Los elementos especificados en un inicializador de arreglo pueden ser expresiones arbitrarias, lo cual difiere de C, donde deben ser expresiones constantes.

En Java 1.1, los arreglos también se crean e inicializan "anónimamente", al combinar la sintaxis `new` con la sintaxis de inicializador. Tal hecho se parece a esto:

```
Menu m = createMenu("Archivo", new String[]{"Abrir...",
"Salvar", "Salir"});
```

Los arreglos se recolectan automáticamente como basura, justo como sucede con los objetos.

Arreglos multidimensionales

Java también soporta los arreglos multidimensionales. Éstos se instrumentan como arreglos de arreglos, como en C. Usted especifica una variable como arreglo de tipo multidimensional con sólo anexar el número adecuado de pares `[]` después de ella. Para asignar un arreglo multidimensional con `new`, usted especifica el número adecuado de elementos (entre corchetes cuadrados) para cada dimensión. Por ejemplo:

```
byte ArregloDosDim[][] = new byte[256][16];
```

Esta declaración hace tres cosas:

- Declara una variable denominada `ArregloDosDim`. Esta variable es del tipo `byte[][]` (arreglo de arreglo de `byte`).
- Asigna dinámicamente un arreglo con 256 elementos. El tipo de este arreglo recién asignado es `byte[][]`, por lo que puede asignarse a la variable que usted declare. Cada elemento de este arreglo nuevo es del tipo `byte[]`, un arreglo unidimensional de `byte`.
- Dinámicamente asigna 256 arreglos de bytes, cada uno de los cuales aloja 16 bytes y almacena una referencia a estos 256 arreglos `byte[]` en los 256 elementos del arreglo `byte[][]`, asignado en el segundo paso. Los 16 bytes de cada uno de estos 256 arreglos se inicializan en su valor por omisión de 0.

Cuando usted asigna un arreglo multidimensional, no tiene que especificar el número de elementos contenidos en cada dimensión. Por ejemplo:

```
int tresD[][][] = new int[10][][];
```

Esta expresión asigna un arreglo que contiene diez elementos, cada uno del tipo `int[][]`. Se trata de una asignación unidimensional, aunque cuando los elementos del arreglo se inicialicen en valores significativos, el arreglo será multidimensional. La regla para esta clase de asignación de arreglo es que las primeras n dimensiones (donde n es por lo menos uno) tengan especificado el número de elementos, y que estas dimensiones vayan seguidas por m dimensiones adicionales sin tamaño de dimensión especificado. Lo que sigue a continuación es válido:

```
String cadenas_perdidas[][][] = new String[5][3][][];
```

Pero esto no:

```
double datos_temperatura[][][] = new double[100][][10]; //no valido
```

Los arreglos multidimensionales también se pueden asignar e inicializar con inicializadores anidados. Por ejemplo, usted podría declarar el siguiente arreglo multidimensional de cadenas para que las usara el método `conseguirInfoParametros()` de un applet:

```
String info_param[][] = {
    {"foreground", "Color", "color de primer plano"},
    {"background", "Color", "color de fondo"},
    {"message", "String", "el banner a mostrar"}
};
```

Observe que, debido a que Java instrumenta arreglos multidimensionales como arreglos de arreglos, los arreglos multidimensionales no necesitan ser "rectangulares". Por ejemplo, he aquí la manera como usted podría crear e inicializar un "arreglo triangular":

```
short triangulo[][] = new short[10][]; // Un arreglo simple
for (int i = 0; i < triangulo.length; i++) { // para cada elemento
    // del arreglo
    triangulo[i] = new short[i+1]; // Aloja un nuevo arreglo
    for (int j = 0; j < i+1; j++) // para cada elemento
        // del nuevo arreglo.
        triangulo[i][j] = (short) i+j; // Lo inicializa con un valor
}
```

Usted también podría declarar e inicializar los arreglos no rectangulares con inicializadores anidados:

```
static int[][] dosdim = {{1, 2}, {3, 4, 5}, {5, 6, 7, 8}};
```

Para simular dimensiones múltiples en un arreglo unidimensional, usted usaría código, justo como lo haría en C:

```
final int renglones = 600;
final int columnas = 800;
byte pixeles[] = new byte[renglones*columnas];

// el acceso a los elementos [i,j] es así:
byte b = pixeles[i+j*columnas];
```

Cómo tener acceso a elementos de arreglo

El acceso a arreglo en Java es como el acceso a arreglo en C: usted tiene acceso a un elemento de un arreglo al colocar una expresión valuada en entero entre corchetes cuadrados, después del nombre del arreglo.

```
int a[] = new int[100];
a[0] = 0;
for (int i = 1; i < a.length; i++) a[i] = 1 + a[i-1];
```

Observe cómo se computó el número de elementos del arreglo en este ejemplo: al acceder al campo `length` del arreglo. Éste es el único campo que soporta los arreglos. Vea que es un campo (final) constante; es decir, todo intento por almacenar un valor en el campo `length` de un arreglo fracasará.

En todas las referencias de arreglo de Java, el índice se verifica para asegurar que no sea demasiado pequeño (menos de cero) o muy grande (mayor que o igual a la longitud del arreglo). Si el índice está fuera de límites, se lanza un arreglo `ArrayIndexOutOfBoundsException`.^{*} Ésta es otra manera en que Java funciona para evitar defectos (y problemas de seguridad).

¿Los objetos son arreglos?

Es conveniente considerar que los arreglos no son objetos, sino un tipo separado de referencia. Sin embargo, en ciertos casos los arreglos se comportan como objetos. Como ya se vio, los arreglos usan la sintaxis de objeto `.length` para referirse a su longitud. Los arreglos también se pueden asignar a variables del tipo `Object`, y los métodos de la clase `Object` se pueden invocar para los arreglos. (`Object` es la clase raíz en Java, lo cual quiere decir que todos los objetos se pueden asignar a una variable del tipo `Object` e invocar los métodos de `Object`.)

La evidencia sugiere que los arreglos son, de hecho, objetos. Sin embargo, Java define suficiente sintaxis especial para los arreglos, por lo que resulta más adecuado considerarlos como una especie diferente de tipo de referencia, y no objetos.

Declaración de variables y argumentos de arreglo

En C, usted declara una variable de arreglo o argumento de función de arreglo al colocar corchetes cuadrados junto al nombre de la variable:

```
void reversa(char strbuf[], int tamano_buffer) {
    char buffer[500];
    ...
}
```

En Java, usted tendría que declarar `buffer` como una variable de arreglo, para después asignar el arreglo en sí con `new`; de lo contrario, podría usar la misma sintaxis, con los corchetes del arreglo después del nombre de variable o argumento.

Sin embargo, Java también permite que usted ponga los corchetes del arreglo después del nombre de tipo. Por ello usted podría reescribir este fragmento de código para que se pareciera a esto:

```
void reversa(char[] strbuf, int tamano_buffer) {
    char[] buffer = new char[500];
    ...
}
```

De muchas formas, esta nueva sintaxis de arreglo es más fácil de leer y de entenderse (por cierto, no funciona en C, ya que los punteros hacen que la sintaxis de declaración de tipo en C sea un verdadero embrollo). El único problema con esta sintaxis nueva es que si usted se acostumbra a usarla, le será más difícil volver a programar en C, ¡cosa que ojalá sólo suceda ocasionalmente!

^{*} Más adelante, en este capítulo, se describen las excepciones y el manejo de excepción.

Java incluso permite mezclar los estilos de declaración, lo cual a veces le parecerá útil (¡o frecuentemente confuso!) para ciertas estructuras de datos o algoritmos. Por ejemplo:

```
// renglon y columna son arreglos de byte
// matriz es un arreglo de un arreglo de byte
byte[] renglon, columna, matriz[];

// Este metodo toma un arreglo de bytes y un
// arreglo de arreglos de bytes
public void dot_producto(byte[] columna, byte[] matriz[]) { ... }
```

Una última nota respecto a las declaraciones de arreglo es que (como se ha visto en toda esta sección) el tamaño de un arreglo no es parte de su tipo, como en C. Por ello, usted puede declarar una variable para que sea del tipo `String[]`, por ejemplo, y asignar a ella cualquier arreglo de objetos `String`, sin importar la longitud del arreglo:

```
String[] cadenas;           // esta variable puede referir cualquier
                             // arreglo String
cadenas = new String[10];   // una que contenga 10 Strings
cadenas = new String[20];   // o una que contenga 20.
```

Cadenas

Las cadenas en Java *no* son arreglos de caracteres terminados en valor nulo, como en C, sino son instancias de la clase `java.lang.String`. Las cadenas de Java no son normales, ya que el compilador las trata casi como si fueran tipos primitivos, por ejemplo, y éste crea automáticamente un objeto `String` cuando encuentra una constante entre comillas dobles, en el programa. Además, el lenguaje define un operador que opera sobre los objetos `String`; éste es el operador para la concatenación de cadena.

Una característica importante de los objetos `String` es que son inmutables; es decir, no hay métodos definidos que permitan cambiar los contenidos de una `String`, por lo que si usted necesita modificarlos, tiene que crear un objeto `StringBuffer` desde el objeto `String`, modificar los contenidos del `StringBuffer` y después crear una nueva `String` desde los contenidos del `StringBuffer`.

Resulta controvertido contestar la pregunta de si las cadenas de Java terminan o no con un carácter NUL (`\u0000`). Java efectúa verificaciones de límites al tiempo de ejecución en todos los accesos de arreglo y cadena, así que no hay manera de examinar el valor de algún carácter terminador interno que aparezca después del último carácter de la cadena.

Las clases `String` y `StringBuffer` se documentan en el capítulo 25, donde usted encontrará un conjunto completo de métodos para manipular y manejar las cadenas. Algunos de los métodos `String` más importantes son: `length()`, `charAt()`, `equals()`, `compareTo()`, `indexOf()`, `lastIndexOf()` y `substring()`.

Operadores

Java soporta casi todos los operadores estándares de C. Estos operadores estándares tienen la misma precedencia y asociabilidad tanto en Java como en C. Se listan en la tabla 2-3 y también en la tabla de referencia rápida del capítulo 13.

Tabla 2-3: operadores de Java

Prec.	Operador	Tipo(s) de operando	Asoc.	Operación efectuada
1	++	aritmético	D	pre o post incremento (unario)
	--	aritmético	D	pre o post decremento (unario)
	+, -	aritmético	D	unario más, unario menos
	~	entero	D	complemento bitwise (unario)
	!	booleano	D	complemento lógico (unario)
2	(tipo)	cualquiera	D	lanzar
	*, /, %	aritmético	I	multiplicación, división, resta
3	+, -	aritmético	I	suma, resta
	+	cadena	I	concatenación de cadena
4	<<	entero	I	movimiento a la izquierda
	>>	entero	I	movimiento a la derecha con extensión de signo
	>>>	entero	I	movimiento a la derecha con extensión cero
5	<, <=	aritmético	I	menor que, menor que o igual
	>, >=	aritmético	I	mayor que, mayor que o igual
6	instanceof	objeto, tipo	I	comparación de tipo
	==	primitivo	I	igual (tiene valores idénticos)
	!=	primitivo	I	no igual (tiene valores diferentes)
	==	objeto	I	igual (referencia al mismo objeto)
	!=	objeto	I	no igual (referencia a objetos diferentes)
7	&	entero	I	AND bitwise
	&	booleano	I	AND booleano
8	^	entero	I	XOR bitwise
	^	booleano	I	XOR booleano
9		entero	I	OR bitwise
		booleano	I	OR booleano
10	&&	booleano	I	AND condicional
11		booleano	I	OR condicional
12	?:	booleano, cualquiera	D	operador condicional (ternario)
13	=	variable, cualquiera	D	asignación
	*, /=, %=, +=, -=,	variable, cualquiera	D	asignación con operación
	<<=, >>=,			
	>>>=, &=,			
	^=, =			

Observe las siguientes diferencias entre el operador de Java y C. Java no soporta el operador de coma para combinar dos expresiones en una sola (aunque la declaración `for` simula este operador en forma útil). Como Java no permite que usted manipule los

punteros directamente, no soporta los operadores de referencia y desreferencia *, -> y &, ni el operador sizeof. Por otra parte, Java no considera [] (acceso a arreglo) y . (acceso a campo) como operadores, a diferencia de C.

Java también agrega algunos operadores nuevos:

+

El operador + aplicado a los valores String los concatena.* Si sólo un operando de + es una String, el otro se convierte en una cadena. La conversión es automática en el caso de los tipos primitivos, y al llamar el método toString() de tipos no primitivos. Este operador String + tiene la misma precedencia que el operador aritmético +. El operador += funciona como usted lo esperaría con los valores String.

instanceof

El operador instanceof regresa true si el objeto o localizado en su costado izquierdo es una instancia de la clase C o instrumenta la interfaz I especificada en su costado derecho. También regresa true si o es una instancia de una subclase de C o es una instancia de una subclase de alguna clase que instrumente I. instanceof regresa false si o no es una instancia de C o no instrumenta I. También regresa false si el valor ubicado a su izquierda es null. Si instanceof regresa true, quiere decir que o es assignable a variables del tipo C o I. El operador instanceof tiene la misma precedencia que los operadores <, <=, > y >=.

>>>

Debido a que todos los tipos enteros en Java son valores con signo, el operador >> de Java se define para hacer un cambio a la derecha con la extensión de signo. El operador >>> trata el valor para que cambie como número sin signo y cambia los bits a la derecha con extensión cero. El operador >>>= funciona como usted lo esperaría.

& y |

Cuando & y | se aplican a tipos enteros en Java, efectúan las operaciones bitwise AND y OR esperadas. Sin embargo, Java hace una marcada distinción entre los tipos enteros y el tipo boolean. Por ello, si estos operadores se aplican a tipos boolean, efectúan operaciones lógicas AND y OR. Estos operadores lógicos AND y OR siempre evalúan sus operandos, incluso cuando el resultado de la operación es determinado después de evaluar solamente el operando izquierdo. Lo anterior resulta útil cuando los operandos son expresiones con efectos laterales (por ejemplo, llamadas de método) y usted quiere que ocurran estos efectos. Sin embargo, si usted no desea que el operador de la derecha sea evaluado porque no es necesario, puede usar los operadores && y ||, los cuales efectúan operaciones lógicas AND y OR “de circuito corto”, justo como en C. Los operadores &= y |= realizan una operación bitwise o lógica, dependiendo del tipo de operandos, como uno habría de esperarlo.

* Para los programadores en C++, esto parece una sobrecarga de operador. De hecho, Java no respalda el sobrecargado de operador; en otras palabras, los diseñadores del lenguaje decidieron (después de mucho debatir) que los operadores sobrecargados eran una idea brillante, pero que se volvía difícil de leer y entender el código del cual dependían.

Declaraciones

Muchas declaraciones de control de Java son similares o idénticas a las declaraciones de C. En esta sección se listan y explican, según sea necesario, las declaraciones de Java. El tema de las excepciones y la declaración `try/catch/finally` es bastante sustancial y por ello se aborda en una sección posterior.

Las declaraciones *if/else*, *while* y *do/while*

Las declaraciones `if`, `else`, `do` y `while` son exactamente las mismas tanto en Java como en C. La única diferencia sustancial es que el tipo `boolean` de Java no puede lanzarse a otros tipos. En Java, los valores `0` y `null` no son lo mismo que `false`, y los valores `no cero` y `no null` no son lo mismo que `true`.

La expresión condicional que esperan las declaraciones `if`, `while` y `do/while` debe ser del tipo `boolean` en Java. Especificar un tipo entero o uno de referencia no será suficiente. Por ello, el siguiente código de C no es válido en Java:

```
int i = 10;
while (i--) {
    Object o = conseguir_objetos();
    if (0) {
        do { ... } while (j);
    }
}
```

En Java, usted debe hacer clara la condición que habrá de probar. Para lograrlo, pruebe explícitamente el valor contra `0` o `null`. Use un código como el siguiente:

```
int i = 10;
while (i-->0) {
    Object o = conseguir_objetos();
    if (0 != null) {
        do { ... } while (j);
    }
}
```

La declaración *switch*

La declaración `switch` es la misma en Java y en C. Usted puede usar tipos `byte`, `char`, `short`, `int` o `long` como los valores de las etiquetas `case`, además de especificar una etiqueta `default`, tal como se puede hacer en C.

El ciclo *for*

La declaración `for` es quizá el constructor de ciclo más útil en Java. Sólo hay dos diferencias entre el ciclo `for` de Java y el ciclo `for` de C. La primera diferencia es que, aunque Java no soporta el operador de coma de C (que permite enlazar múltiples expresiones en una sola expresión), el ciclo `for` de Java lo simula al permitir que múltiples expresiones separadas por coma aparezcan en las secciones de inicialización e incremento, pero no en la sección de prueba, del ciclo. Por ejemplo:

```
int i;
string s;
for (i=0, s="probando";           // Inicializa variables.
    (i < 10) && (s.length() >= 1); // Prueba para continuar.
```

```

    i++, s = s.substring(1))           // Incrementa variables.
{
    System.out.println(s);           // Cuerpo del ciclo.
}

```

Como se ve, esta “diferencia” entre los ciclos de Java y C en realidad es una similitud.

La segunda diferencia es la adición de la capacidad de C++ para declarar variables de ciclo local en la sección de inicialización del ciclo:

```

for(int i = 0, i < mi_arreglo.length; i++)
    System.out.println("a[" + i + "] = " + mi_arreglo[i]);

```

Las variables declaradas de esta manera tienen el ciclo `for` como alcance. En otras palabras, sólo son válidas dentro del cuerpo del ciclo `for` y dentro de las expresiones de inicialización, prueba e incremento del ciclo. Aunque las variables declaradas en los ciclos `for` tienen su propio alcance, el compilador de Java no le permite a usted declarar una variable de ciclo que tenga el mismo nombre que una variable o un parámetro existente.

Observe que, como la sintaxis de declaración de variable también usa la coma, la sintaxis de Java le permite a usted especificar múltiples expresiones de inicialización separadas con coma, o declarar e inicializar variables múltiples separadas con coma del mismo tipo. Usted no puede mezclar declaraciones de variable con otras expresiones que no sean de declaración. Por ejemplo, el ciclo `for` siguiente declara e inicializa dos variables válidas solamente dentro del ciclo `for`.

```

for(int i=0, j=10; i < j; i++, j--) System.out.println("k = " + i*j);

```

Declaraciones etiquetadas break y continue

Las declaraciones `break` y `continue`, empleadas a solas, se comportan en Java como lo hacen en C. Sin embargo, en Java pueden ir seguidas opcionalmente por una etiqueta que especifique un ciclo de cierre (para `continue`) o cualquier declaración de cierre (para `break`). Las formas etiquetadas de estas declaraciones permiten que usted “rompa” y “continúe” cualquier declaración o ciclo especificado dentro de una definición de método, y no sólo la declaración o el ciclo de cierre más cercano.

La declaración `break` sin etiqueta, transfiere o quita el control (“la deja sin” o lo termina) de la declaración de cierre `for`, `while`, `do` o `switch` más cercana, justo como en C. Si la palabra clave `break` va seguida de algún identificador que sea la etiqueta de una declaración de cierre arbitraria, la ejecución cesa en dicha declaración. Después de ejecutarse la declaración `break`, se ejecutan todas las cláusulas `finally` requeridas, y el control continúa en la declaración que venga después de la declaración terminada. (La cláusula `finally` y la declaración `try` a que esté asociada, son construcciones de manejo de excepción y se explican en la sección siguiente.) Por ejemplo:

```

test: if (check(i)) {
    try {
        for (int j=0; j < 10; j++) {
            if (j > i) break;           // Termina solo este loop.
            if (a[i][j] == null)
                break test;           // Ejecuta la clausula Finally y
                                     // termina la sentencia if
        }
    } finally { cleanup(a, i, j); }
}

```

Sin una etiqueta, la declaración `continue` funciona exactamente como en C: detiene la iteración en progreso y ocasiona que se reasuma la ejecución después de la última declaración en el ciclo `while`, `do` o `for`, justo antes de que comience otra vez la iteración. Si la palabra clave `continue` va seguida por un identificador que sea la etiqueta de un ciclo de cierre, esta vez la ejecución salta al final de dicho ciclo. Si hay alguna cláusula `finally` entre la declaración `continue` y el final del ciclo adecuado, estas cláusulas se ejecutan antes de que el control se transfiera al final del ciclo.

El siguiente fragmento de código ilustra la manera en que funciona la declaración `continue` en sus formas etiquetada y sin etiqueta.

```
gran_loop: while (!hecho) {
    if (test(a,b) == 0) continue;    // El control va al punto 2.
    try {
        for(int i=0; i < 10; i++) {
            if (a[i] == null)
                continue;           // El control va al punto 1.
            else if (b[i] == null)
                continue gran_loop;  // El control va al punto 2,
                                    // después de ejecutar
                                    // el bloque finally.
            doit(a[i],b[i]);
            // Punto 1. Incrementa e inicia el loop otra vez con el test.
        }
    } finally { cleanup(a,b); }
    // Punto 2. Inicia el loop otra vez con el test (!hecho).
}
```

Observe la característica no intuitiva de la declaración `continue` etiquetada: la etiqueta de ciclo debe aparecer al comienzo del ciclo, pero `continue` ocasiona que la ejecución se transfiera hacia la parte más inferior del ciclo.

Sin declaración goto

`goto` es una palabra reservada en Java, pero la declaración `goto` en realidad no es parte del lenguaje. Las declaraciones con etiqueta `break` y `continue` reemplazan algunos usos importantes y legítimos de `goto`, mientras la declaración `try/catch/finally` reemplaza a los otros.

La declaración synchronized

Como Java es un sistema de hilos múltiples, debe tenerse cuidado para evitar que los hilos múltiples modifiquen objetos simultáneamente, en una forma que pudiera corromper el estado de los objetos. Las secciones de código que no deben ejecutarse simultáneamente se conocen como "secciones críticas". Java proporciona la declaración `synchronized` para proteger estas secciones. La sintaxis es:

`synchronized (expresión) sentencia`

expresión es una expresión que debe resolver a un objeto o arreglo. La *statement* es el código de la sección crítica, el cual es por lo general un bloque de declaraciones (dentro de `{ y }`). La declaración `synchronized` intenta adquirir un candado exclusivo para el objeto o arreglo especificado por *expresión*; no ejecuta la sección crítica de código sino hasta que puede obtener este candado y, de esta manera, asegura que ningún otro hilo ejecute la sección al mismo tiempo.

Usted no tiene que usar la declaración `synchronized`, a menos que su programa cree múltiples hilos que compartan datos. Si sólo un hilo accesa una estructura de datos, no hay necesidad de protegerla con `synchronized`. Cuando usted tenga que usarla, podría hacerlo en un código como el siguiente:

```
public static void ArregloOrdenado(int[] a) {
    // Ordenar el arreglo a. Este es synchronized así que otros
    // hilos no pueden cambiar elementos del arreglo mientras
    // lo estamos ordenando.
    // Al menos no otros hilos que protejan sus cambios al
    // arreglo con synchronized.
    synchronized (a) {
        // ordena el arreglo aquí.
    }
}
```

La palabra clave `synchronized` a menudo se usa más como modificador de método en Java. Cuando se aplica a un método, indica que todo el método es una sección crítica. En el caso de un método de clase `synchronized` (un método estático), Java obtiene un candado exclusivo sobre la clase antes de ejecutar el método. En el caso de un método de instancia `synchronized`, Java obtiene un candado exclusivo sobre la instancia de clase. (Los métodos de clase y de instancia se describen en el capítulo siguiente.)

Las declaraciones `package` e `import`

La declaración `package`, como se vio con anterioridad en este capítulo, especifica el paquete del que son parte las clases de un archivo de código fuente de Java. En caso de aparecer, debe ser la primera declaración en un archivo de Java. La declaración `import`, también explicada con anterioridad, permite que usted haga referencia a clases mediante nombres abreviados. Las declaraciones `import` deben aparecer después de la declaración `package`, de haberlas, y antes de cualquier otra declaración en un archivo de Java. Por ejemplo:

```
package juegos.tetris;

import java.applet.*;
import java.awt.*;
```

Excepciones y manejo de excepción

El manejo de excepción es una característica nueva e importante de Java.* Existen términos nuevos asociados con el manejo de excepción. En primer lugar, una *excepción* es una señal de que ha ocurrido cierto tipo de condición excepcional (por ejemplo, un error). *Lanzar* una excepción expresa una condición excepcional. *Atrapar* una excepción equivale a manejarla; es decir, tomar todas las acciones necesarias para recuperarse de ella.

Las excepciones se propagan por la estructura de bloque léxico de un método de Java, y después por la pila de llamado de método. Si una excepción no es atrapada por el bloque de código que la lanzó, ésta se propaga al siguiente bloque de cierre de código superior. Si no se atrapa ahí, se vuelve a propagar. De no ser atrapada en ningún sitio del método, se propaga al método de llamada, donde vuelve a propagarse por la estructura de bloque. Si una excepción nunca es atrapada, se propaga por todo el método `main()` desde el cual se haya iniciado el programa, y ocasiona que el intérprete de Java imprima un mensaje de error y una traza de pila, para después salir.

* Aunque es similar, no es igual al manejo de excepción en C++.

Como se verá en las subsecciones siguientes, las excepciones hacen que el manejo de error (y el manejo de “condición excepcional”) sea más regular y lógico, pues permite que usted agrupe todo el código de manejo de excepción en un solo lugar. Así, en vez de preocuparse por todo lo que pueda salir mal en cada línea del código, usted puede concentrarse en el algoritmo a la mano y colocar todo el código de manejo de error (esto es, el código para atrapar excepciones) en un solo sitio.

Objetos de excepción

Una excepción en Java es un objeto que es instancia de alguna subclase de `java.lang.Throwable`. `Throwable` tiene dos subclases estándares: `java.lang.Error` y `java.lang.Exception`.^{*} Las excepciones que son subclases de `Error` generalmente indican problemas de liga relacionados con el cargado dinámico, o problemas de la máquina virtual, como el agotamiento de memoria. Casi siempre deben considerarse como irrecuperables, y no deben ser atrapadas. Aunque la distinción no es clara en todo momento, las excepciones que sean subclases de `Exception` expresan condiciones que se pueden atrapar y de las cuales es posible recuperarse. Entre ellas se incluyen excepciones como `java.io.EOFException`, que marca el final de un archivo, y `java.lang.ArrayAccessOutOfBoundsException`, que indica que un programa ha tratado de leer más allá del final de un arreglo.

Como las excepciones son objetos, pueden contener datos y definir métodos. El objeto `Throwable`, al comienzo de la jerarquía de clase de excepción, incluye un mensaje `String` en su definición y este campo es heredado por todas las clases de excepción. Dicho campo se usa para almacenar un mensaje de error legible para el humano, donde se describe la condición excepcional; se configura cuando el objeto de excepción se crea al pasar un argumento al método constructor. El mensaje se puede leer desde la excepción con el método `Throwable.getMessage()`. La mayoría de las excepciones contiene solamente este mensaje, pero algunas agregan más datos. `java.io.InterruptedIOException`, por ejemplo, agrega el campo siguiente:

```
public int bytesTransferidos;
```

Este campo especifica cuánto de la E/S se completó antes de que ocurriera la condición excepcional.

Manejo de excepción

La declaración `try/catch/finally` es el mecanismo de manejo de excepción de Java. `try` establece un bloque de código que manejará las excepciones y salidas anormales (vía `break`, `continue`, `return` o la propagación de excepción). El bloque `try` va seguido por cero o más cláusulas `catch` que atrapan y manejan los tipos especificados de excepciones. Las cláusulas `catch` opcionalmente van seguidas de un bloque `finally` que contiene código “de limpieza”. Las declaraciones de un bloque `finally` se ejecutarán con garantía, sin importar cómo salga el código en el bloque `try`. En el ejemplo 2-2 se muestra la sintaxis detallada de `try/catch/finally`.

Ejemplo 2-2: La declaración try/catch/finally

```
try {  
    // Normalmente este código corre desde arriba del bloque hasta  
    // el final sin problemas. Pero algunas veces puede ocasionar  
    // excepciones o salir del bloque con una sentencia break,  
    // continue, o return.
```

^{*} El término “excepción” se usará para referirse a cualquier subclase de `Throwable`, sin importar si realmente es una `Exception` o un `Error`.

Ejemplo 2-2: La declaración try/catch/finally (continuación)

```
}
catch (AlgunaExcepcion e1) {
    // El manejo de un objeto de excepcion e1 de tipo AlgunExcepcion
    // o de una subclase de ese tipo.
}
catch (OtraExcepcion e2) {
    // El manejo de un objeto de excepcion e2 de tipo OtraExcepcion
    // o de una subclase de ese tipo.
}
finally {
    // Siempre ejecuta este codigo, despues de que salimos de la
    // sentencia try, indiferente de como hayamos salido:
    // 1) Normalmente, despues de llegar al final del bloque.
    // 2) Con una excepcion que sea manejada por un catch.
    // 3) Con una excepcion que no sea manejada.
    // 4) A causa de una sentencia break, continue, o return.
}
```

try (tratar)

La cláusula **try** simplemente establece un bloque de código que habrá de manejar todas las excepciones y salidas anormales (vía **break**, **continue**, **return** o propagación de excepción). La cláusula **try**, por sí misma, no hace nada interesante; son las cláusulas **catch** y **finally** las que manejan las excepciones y realizan las operaciones de limpieza.

catch (atrapar)

Un bloque **try** puede ir seguido de cero o más cláusulas **catch**, las cuales especifican el código que manejará los distintos tipos de excepciones. Las cláusulas **catch** tienen una sintaxis inusual: cada una se declara con un argumento, parecido a un argumento de método. Este argumento debe ser del tipo **Throwable** o una subclase. Cuando ocurre una excepción, se invoca la primera cláusula **catch** que tenga un argumento del tipo adecuado. El tipo de argumento debe concordar con el tipo de objeto de excepción, o ser una superclase de la excepción. Este argumento **catch** es válido sólo dentro del bloque **catch**, y hace referencia al objeto de excepción real que fue lanzado.

El código de un bloque **catch** debe tomar las acciones necesarias para enfrentar la condición excepcional. Si la excepción fue **java.io.FileNotFoundException**, por ejemplo, usted podría manejarla al pedir al usuario que verifique la grafía y lo intente de nuevo. No se requiere tener una cláusula **catch** para cada posible excepción (en algunos casos, la respuesta correcta consiste en permitir que la excepción se propague y sea atrapada por el método invocador). En otros casos, como cuando ocurre un error de programación señalado por **NullPointerException**, la respuesta correcta es no atrapar la excepción en absoluto, sino permitir que se propague y hacer que el intérprete de Java salga con una traza de pila y un mensaje de error.

finally (finalmente)

La cláusula **finally** se usa en general para limpiar (cerrar archivos, liberar recursos, etc.) después de la cláusula **try**. Lo útil de la cláusula **finally** es que garantiza la ejecución del código de un bloque **finally**, en caso de que alguna porción del bloque **try** se ejecutara, sin importar cómo se complete el código en el bloque **try**. En un caso

normal, el control llega al final del bloque `try` y posteriormente va hacia el bloque `finally`, que efectúa toda la limpieza necesaria.

Si el control abandona al bloque `try`, debido a una declaración `return`, `continue` o `break`, los contenidos del bloque `finally` se ejecutan antes de que el control se transfiera hacia su nuevo destino.

Si ocurre una excepción en el bloque `try` y hay un bloque `catch` local para manejar la excepción, el control se transfiere primero hacia el bloque `catch` y después al bloque `finally`. Si no hay un bloque `catch` local para manejar la excepción, el control se transfiere primero al bloque `finally` y después se propaga a la cláusula `catch` más cercana que pueda manejar la excepción.

Si un bloque `finally` en sí transfiere el control con una declaración `return`, `continue` o `break`, o al emitir una excepción, la transferencia pendiente de control se abandona y se procesa esta nueva transferencia.

Por otra parte, `try` y `finally` se pueden usar entre sí, sin excepciones o cláusulas `catch`. En este caso, el bloque `finally` simplemente es código de limpieza que sin duda se ejecutará, sin importar las declaraciones `break`, `continue` o `return` de la cláusula `try`.

Declaración de excepciones

Java requiere que todo método que pueda ocasionar una “excepción normal”, atrape la excepción o especifique el tipo de excepción con una cláusula `throws` en la declaración del método.* La cláusula `throws` puede verse así:

```
public void abrir_archivo () throws IOException {  
    // Las sentencias aquí deben generar un java.io.IOException  
}  
  
public void mifunc(int arg) throws MiExcepción1, MiExcepción2 {  
    ...  
}
```

Observe que la clase de excepción especificada en una cláusula `throws` puede ser una superclase del tipo de excepción que se lance en realidad. Así, si un método lanza excepciones `a`, `b` y `c`, todas ellas subclases de `d`, la cláusula `throws` podría especificar `a`, `b` y `c`, o simplemente especificar `d`.

Se dice que la cláusula `throws` se debe usar para declarar “excepciones normales”. Esta frase oximorónica se refiere a cualquier subclase de `Throwable` que no sea una subclase de `Error` o de `RuntimeException`. Java no requiere que estos tipos de excepción se declaren, pues prácticamente cualquier método puede generarlas y sería tedioso declararlas todas. Por ejemplo, cualquier método que se ejecute en un intérprete defectuoso de Java puede lanzar una excepción `InternalError` (una subclase de `Error`) y no tiene sentido declarar ésta en una cláusula `throws` por cada método. De manera similar, hasta donde concierne al compilador de Java, cualquier método con acceso a un arreglo puede generar una excepción `ArrayIndexOutOfBoundsException` (una subclase de `RuntimeException`).

* Nota para los programadores de C++: observen que Java usa `throws` en donde C++ usa `throw`.

Las excepciones estándar que usted a menudo debe declarar son `java.io.IOException` y un número de sus subclases más específicas. `java.lang.InterruptedException` y otras excepciones de uso menos común también se deben declarar. ¿Cómo saber cuándo se tiene que declarar una cláusula `throws`? Una manera consiste en fijarse en la documentación para los métodos que usted llame; si se pueden lanzar “excepciones normales”, usted tiene que atraparlas o declararlas. Otra forma de saber qué excepciones se tienen que declarar es no declarar ninguna, esperar que surjan errores de compilación ¡y el compilador le dirá qué poner en la cláusula `throws`!

Definición y generación de excepciones

Usted puede señalar sus propias excepciones con la declaración `throw`. La palabra clave `throw` debe ir seguida por un objeto que sea `Throwable` o una subclase. A menudo, los objetos de excepción se asignan en la misma declaración en que son lanzados:

```
throw new MiExcepcion("mi excepcional condicion ocurrida");
```

Cuando se lanza una excepción, se detiene la ejecución normal del programa y el intérprete busca una cláusula `catch` que pueda manejar la excepción. La ejecución se propaga por las declaraciones de cierre y por las funciones de invocación, hasta encontrar la cláusula manejadora. Todo bloque `finally` que se pase durante esta propagación será ejecutado.

Usar excepciones es una buena manera de señalar y manejar los errores en el código de usted. Al agrupar todo el código para manejo de error y recuperación dentro de la estructura `try/catch/finally`, usted tendrá al final un código de limpieza que es más sencillo de entender. En ocasiones, cuando usted lanza una excepción, puede usar una de las clases de excepción ya definidas por la API de Java. Sin embargo, a veces usted querrá definir y lanzar sus propios tipos de excepción.

En el ejemplo 2-3, se muestra cómo puede usted definir sus propios tipos de excepción, lanzarlos y manejarlos. La tabla también ayuda a establecer cómo se propagan las excepciones. Es un ejemplo largo, pero vale la pena estudiarlo con detenimiento. Si puede responder lo siguiente, entonces usted ya entiende el manejo de excepción: ¿qué pasa cuando este programa se invoca sin argumento, con un argumento de cadena y con argumentos de entero 0, 1, 2 y 99?

Ejemplo 2-3: Definición, lanzamiento y manejo de excepciones

```
// Aqui definimos algunos tipos de excepciones de nosotros.
// Las clases de excepcion generalmente tienen constructores pero no
// datos ni otros metodos. Todas estas llaman a los constructores
// de su superclase.
class MiExcepcion extends Exception {
    public MiExcepcion() { super(); }
    public MiExcepcion(String s) { super(s); }
}
class MiOtraExcepcion extends Exception {
    public MiOtraExcepcion() { super(); }
    public MiOtraExcepcion(String s) { super(s); }
}
class MiSubExcepcion extends MiExcepcion {
    public MiSubExcepcion() { super(); }
    public MiSubExcepcion(String s) { super(s); }
}

public class pruebathrow {
    // Este es el metodo main(). Note que usa dos
    // clausulas catch para manejar dos excepciones estandar de Java.
```

Ejemplo 2-3: Definición, lanzamiento y manejo de excepciones (continuación)

```
public static void main(String argv[]) {
    int i;

    // Primero, convierte nuestro argumento en un entero.
    // Nos aseguramos que tengamos un argumento y que es convertible.
    try { i=Integer.parseInt(argv[0]); }
    catch (ArrayIndexOutOfBoundsException e) { // argv esta vacio
        System.out.println("Debe especificar un argumento");
        return;
    }
    catch (NumberFormatException e) { // argv[0] no es un entero
        System.out.println("Debe especificar un argumento entero");
        return;
    }

    // Ahora, pasamos el entero al metodo a().
    a(i);
}

// Este metodo invoca b(), el cual declara un tipo
// de excepcion. Manejaremos una excepcion.
public static void a(int i) {
    try {
        b(i);
    }
    catch (MiExcepcion e) { // Punto 1
        // Aqui manejamos Miexcepcion y su subclase MiSubExcepcion
        if (e instanceof MiSubExcepcion)
            System.out.print("MiSubExcepcion: ");
        else
            System.out.print("MiExcepcion: ");
        System.out.println(e.getMessage());
        System.out.println("Manejado en el punto 1");
    }
}

// Este metodo invoca c(), y maneja uno de los dos tipos de excepciones
// que el metodo puede lanzar. El otro tipo de excepcion no
// es manejado, y es propagado y declarado en la clausula que lo
// lanza en este metodo. Este metodo tambien tiene una clausula finally
// para terminar el trabajo en su clausula try. Note que la clausula
// finally es ejecutada despues de una clausula catch local, pero antes
// de una clausula catch contenedora o una en procedimiento invocador.
public static void b(int i) throws MiExcepcion {
    int resultado;
    try {
        System.out.print("i = " + i);
        resultado = c(i);
        System.out.println(" c(i) = " + resultado);
    }
    catch (MiOtraExcepcion e) { // Punto 2
        // Manejo de MiOtraExcepcion:
        System.out.println("MiOtraExcepcion: " + e.getMessage());
        System.out.println("Manejada en el punto 2");
    }
    finally {
```

Ejemplo 2-3: Definición, lanzamiento y manejo de excepciones (continuación)

```
// Termina la salida de arriba con una nueva línea.
System.out.print("\n");
}
}

// Este metodo computa un valor o lanza una excepcion.
// La clausula que lanza solo lista dos excepciones, porque
// una de las excepciones lanzadas es una subclase de otra.
public static int c(int i) throws MiExcepcion, MiOtraExcepcion {
    switch (i) {
        case 0: // procesando la continuacion en el punto 1
            throw new MiExcepcion("entrada demasiado baja");
        case 1: // procesando la continuacion en el punto 1
            throw new MiSubExcepcion("la entrada sigue demasiado baja");
        case 99: // procesando la continuacion en el punto 2
            throw new MiOtraExcepcion("entrada demasiado alta");
        default:
            return i*i;
    }
}
}
```

Miscelánea de diferencias

En las secciones siguientes, se describen varias diferencias misceláneas entre Java y C. Las diferencias misceláneas que se hayan mencionado en otras partes (por ejemplo, la ausencia de la declaración `goto` y del operado `sizeof`) no se repiten aquí.

Declaraciones de variable local

Una característica que Java ha tomado prestada de C++ es la capacidad de declarar e inicializar variables locales, desde cualquier punto en un cuerpo de método u otro bloque de código. Las declaraciones y sus inicializadores ya no tienen que ser las primeras declaraciones en un bloque; usted puede declararlas donde sea conveniente para la estructura de su código.

Sin embargo, no permita que esta libertad le reste orden; quien lea su programa, verá con agrado que las declaraciones de variable se han agrupado en un solo sitio. Como regla, ponga las declaraciones en la parte superior del bloque, a menos que por motivos poderosos de organización decida colocarlas en otro sitio.

Referencias adelantadas (forward)

Para la eficiencia del compilador, C requiere que las variables y funciones se definan, o por lo menos se declaren, antes de poderse usar o llamar. Esto es, las referencias adelantadas no se permiten en C. Java no establece esta restricción y al eliminarla, también termina con todo el concepto de una declaración de variable o función que está separada de la definición.

Java acepta referencias adelantadas muy flexibles. Un método puede hacer referencia a una variable u otro método de su clase, sin importar en qué sitio de la clase actual se defina la variable o el método. De manera similar, éste puede hacer referencia a cualquier clase, sin importar el sitio en el archivo actual (o fuera del archivo) en que dicha

clase se haya definido. No se permiten referencias adelantadas únicamente en la inicialización de variable. Un inicializador de variable (para variables locales, de clase o de instancia) no puede hacer referencia a otras variables que no se hayan declarado o inicializado todavía.

Sobrecarga de método

Una técnica que Java toma de C++ se denomina *sobrecarga de método*. Los métodos sobrecargados son los que tienen el mismo nombre, pero distintas señales. En otras palabras, toman distintos tipos de argumentos, un número diferente de argumentos o el mismo tipo de argumentos, pero en distintas posiciones en la lista de argumento. No es posible sobrecargar un método al cambiar solamente su tipo de retorno. Dos métodos con el mismo nombre pueden tener distintos tipos de retorno, pero sólo si los argumentos de método también difieren. De manera similar, dos métodos sobrecargados pueden lanzar excepciones distintas, pero sólo si sus argumentos también difieren.

La sobrecarga de método se usa comúnmente en Java para definir funciones relacionadas con el mismo nombre, pero distintos argumentos. Los métodos sobrecargados por lo general ejecutan la misma operación básica, mas permiten que el programador especifique argumentos de distintas maneras, dependiendo de lo que convenga en una situación dada. La sobrecarga de método se describe con más detalle en el capítulo siguiente.

La palabra clave void (vacío)

La palabra clave `void` se usa en Java, como en C, para indicar que una función no regresa valor. (Como se verá en la sección siguiente, los métodos constructores constituyen una excepción a esta regla.)

Java difiere de C (y es similar a C++) en que los métodos que no usan argumentos se declaran con paréntesis vacíos y no con la palabra clave `void`. Java no tiene ningún tipo `void *` ni usa un lanzador (`void`) para ignorar el resultado regresado por una llamada a un método no `void`.

Modificadores

Java define un número de palabras clave modificadoras que son aplicables a variables y/o declaraciones de método, para proporcionar información adicional o colocar restricciones en la variable o el método:

final

La palabra clave `final` es un modificador aplicable a clases, métodos y variables. Tiene un significado similar, pero no idéntico, en cada caso. Una clase `final` puede no ser nunca una subclase. Un método `final` quizá nunca quede invalidado. Una variable `final` tal vez nunca tenga su valor fijo. En Java 1.1, este modificador también se puede aplicar a variables locales y parámetros de método. Este modificador se describe con mayor detalle en el siguiente capítulo.

native

`native` es un modificador aplicable a declaraciones de método. Indica que el método se instrumenta en cualquier parte de C o en otra plataforma. Un método `native` debe tener un signo de punto y coma en vez de cuerpo.

synchronized

En una sección previa se explicó la palabra clave `synchronized`, como una declaración que marcaba una sección crítica de código; pero esa misma palabra también

se puede usar como un modificador de métodos de clase o instancia. Así, indica que el método modifica el estado interno de la clase o de una instancia de la clase, de manera que no sea a prueba de hilos. Antes de ejecutar un método de clase `synchronized`, Java obtiene un candado en la clase, para asegurar que ningún otro hilo modifique la clase en ese momento. Antes de ejecutar un método de instancia `synchronized`, Java obtiene un candado en la misma instancia que invocó al método, para asegurar que ningún otro hilo modifique el objeto al mismo tiempo.

transient

La palabra clave `transient` es un modificador que se puede aplicar a campos de instancia en una clase. Este modificador es válido, pero no se usa en Java 1.0. En Java 1.1, indica un campo que no es parte del estado persistente de un objeto, y por ello no necesita ponerse en serie con el objeto.

volatile

La palabra clave `volatile` es un modificador que se puede aplicar a campos. Especifica que el campo es usado por hilos sincronizados y que el compilador no debe intentar efectuar optimizaciones con él. Por ejemplo, debe leer cada vez el valor de la variable de memoria y no intentar guardar una copia de éste en la pila.

Sin estructuras ni uniones

Java no soporta los tipos `struct` o `union` de C. Sin embargo, una `class` es en esencia lo mismo que una `struct`, pero con más características. Y usted puede simular las características importantes de una `union` mediante subclases.

Sin tipos numerados

Java no soporta la palabra clave `enum` para definir tipos que consistan en uno de los valores específico nombrados. Esto resulta un poco increíble en un lenguaje de tipografía severa, como lo es Java. Los tipos numerados se pueden simular parcialmente con el uso de valores constantes `static final`.

Sin tipos de método

C permite que usted almacene la dirección de una función en una variable, y que pase las direcciones de funciones a otras funciones. No es posible hacer esto en Java: los métodos no son datos y no se pueden manipular mediante los programas de Java. Sin embargo, los objetos son datos y pueden definir métodos.* Por ello, cuando necesite pasar de un método a otro método, declare una clase que defina el método deseado, y pase una instancia de dicha clase. Observe, por ejemplo, la interfaz `FilenameFilter` en el paquete `java.io`.

Sin campos de bit

Java no soporta la capacidad de C para definir variables que ocupen bits, en particular dentro de los tipos `struct` y `union`. Esta característica de C en general sólo se usa para hacer interfaz directamente con dispositivos de hardware, lo cual nunca es necesario con el modelo de programación independiente de plataforma de Java.

* Una manera interesante de conceptualizar los objetos es pensar que constituyen un tipo de método que define varios puntos de entrada.

Sin typedef (definición de tipo)

Java no soporta la palabra clave `typedef` para definir alias para nombres de tipo; sin embargo, tiene un esquema de nombrado mucho más sencillo que el de C, así que no hay necesidad de tener algo como `typedef`.

Sin listas de argumento de longitud de variable

Java no permite que usted defina métodos que tomen un número variable de argumentos, como lo hace C. Esto se debe a que Java es un lenguaje de tipografía severa y a que no hay manera de efectuar una verificación adecuada de escritura en un método con argumentos variables. La sobrecarga de método permite que usted simule las funciones "varargs" de C en casos simples, pero en general no hay un reemplazo para esta característica de C.



CAPÍTULO 3

Clases y objetos en Java

Java es un lenguaje *orientado a objetos*. "Orientado a objetos" es un término de uso tan común que prácticamente no tiene un significado concreto. En este capítulo se explicará qué significa exactamente para Java estar "orientado a objetos", mediante los temas:

- Clases y objetos en Java
- Creación de objetos
- Recolección de basura para liberarse de objetos no usados
- La diferencia entre variables de clase (o estáticas) y variables de instancia, así como la diferencia entre los métodos de clase (o estáticos) y los métodos de instancia
- Extensión de una clase para crear una subclase
- Invalidación de métodos de clase y búsqueda dinámica de método
- Clases abstractas
- Tipos de interfaz y su instrumentación mediante clases

Si usted es un programador en C++ o tiene alguna experiencia en la programación orientada a objetos, muchos de los conceptos listados aquí le parecerán familiares. Pero si no tiene experiencia con la programación orientada a objetos, no tema: en este capítulo se da por hecho que usted desconoce el tema.

En el capítulo anterior se vieron analogías certeras entre Java y C. Pero desafortunadamente, no se puede decir lo mismo entre Java y C++ para los programadores en C++. Aun así, Java usa conceptos de programación orientada a objetos que resultan familiares para los programadores en C++, incluso toma prestados varios lugares de la sintaxis de C++; pero las analogías entre Java y C++ no son tan claras como las existentes entre Java y C.* Por ello, aunque a los programadores en C++ les parezca más sencillo este capítulo que a los programadores en C, deben leerlo con cuidado y no formarse falsas ideas acerca de Java, con base en su conocimiento de C++.

* Como se verá, Java soporta la recolección de basura y la búsqueda dinámica de método. Esto en realidad lo vuelve un pariente más cercano de lenguajes como Smalltalk, y no de C++ (a pesar de tener una sintaxis parecida a C).

Introducción a las clases y los objetos

Una *clase* es una colección de datos y métodos que operan sobre esos datos.* Los datos y los métodos, en conjunto, generalmente sirven para definir los contenidos y las capacidades de algún tipo de *objeto*.

Por ejemplo, un círculo se puede describir mediante la posición x , y de su centro y por su radio. Se pueden hacer varias cosas con círculos: computar su circunferencia, computar su área, verificar si ciertos puntos están dentro de ellos y así sucesivamente. Cada círculo es diferente (es decir, tiene un centro o radio distinto), pero como *clase*, los círculos tienen ciertas propiedades intrínsecas que se pueden capturar en una definición. El ejemplo 3-1 muestra la manera en que es posible definir parcialmente la clase de círculos en Java. Observe que la definición de clase contiene datos y métodos (procedimientos) dentro del mismo par de llaves.†

Ejemplo 3-1: La clase Círculo, parcialmente capturada en código de Java

```
public class Círculo {  
    public double x, y;    // Las coordenadas del centro  
    public double r;      // El radio  
  
    // El metodo que retorna la circunferencia y el area del circulo  
    public double circunferencia () { return 2 * 3.14159 * r; }  
    public double area () { return 3.14159 * r * r; }  
}
```

Los objetos son instancias de una clase

Ahora que se ha definido la clase *Círculo* (por lo menos parcialmente), será necesario hacer algo con ella. Sin embargo, no se puede hacer nada con la clase *Círculo* en sí, pues se necesita un círculo en particular con que trabajar; se requiere una *instancia* de la clase; es decir, un objeto de círculo.

Al definir la clase *Círculo* en Java, se ha creado un nuevo tipo de dato. Las variables de este tipo se pueden declarar:

```
Circle c;
```

Pero esta variable *c* es simplemente un nombre que *hace referencia* a un objeto de círculo, y no es un objeto en sí. En Java, todos los objetos se deben crear dinámicamente. Esto casi siempre se hace con la palabra clave *new*:

```
Circle c;  
c = new Circle();
```

Ahora se ha creado una instancia de la clase *Círculo* (un objeto de círculo) y se ha asignado a la variable *c*, que es del tipo *Círculo*.

* Un *método* es el término orientado a objetos para un procedimiento o una función. Aparecerá mucho en este libro. Trátele como sinónimo de "procedimiento".

† Los programadores en C++ deben observar que, en Java, los métodos van dentro de la definición de clase y no fuera con el operador `::`, como es el caso en C++.

Cómo tener acceso a los datos del objeto

Ahora que se ha creado un objeto, se pueden usar sus campos de datos. La sintaxis será familiar para los programadores en C:

```
Circle c = new Circle();
c.x = 2.0; // Inicializa nuestro círculo con un centro (2, 2) y radio 1.0
c.y = 2.0;
c.r = 1.0;
```

El uso de métodos de objeto

¡Aquí es donde las cosas comienzan a ponerse interesantes! Para tener acceso a los métodos de un objeto, se usa la misma sintaxis empleada para acceso a los datos de un objeto:

```
Circle c = new Circle();
double a;
c.r = 2.5;
a = c.area();
```

Observe la última línea. No se dice:

```
a = area(c);
```

Sino:

```
a = c.area();
```

Por esto se llama programación "orientada a objetos"; donde el objeto es el foco, no la llamada de función. Quizás ésta sea la característica individual más importante del paradigma de la orientación a objetos.

Vea que no se pasa un argumento a `c.area()`. El objeto sobre el que ahora se opera, `c`, está implícito en la sintaxis. Examine el ejemplo 3-1 nuevamente: observará lo mismo en la definición del método `area()`, es decir, no toma un argumento. Está implícito en el lenguaje que un método opera sobre una instancia de la clase dentro de la cual está definido. Por ello, el método `area()` puede usar con libertad el campo `r` de la clase, el cual se entiende hace referencia al radio de cualquier instancia `Círculo` que invoque al método.

Cómo funciona esto

¿Qué pasa aquí? ¿Cómo es posible que un método que no toma argumentos sepa sobre qué datos opera? Bueno, de hecho, el método `area()` ¡sí tiene un argumento! `area()` se instrumenta con un argumento implícito que no se muestra en la declaración de método. El argumento implícito llama `this` y hace referencia a "este objeto", el objeto `Círculo` mediante el cual se invoca el método. `this` a menudo se denomina como el "this pointer" (este puntero).*

El argumento implícito `this` no se muestra en las señales de método, ya que por lo general esto no es necesario. Cada vez que un método de Java tiene acceso a los campos de su clase, accesa de manera implícita los campos del objeto al que se hace referencia mediante el argumento `this`. Lo mismo sucede, como se verá, cuando un método invoca en una clase a otros métodos en la clase; es decir, está implícito que los métodos se invocan mediante el objeto `this`.

*"this pointer" es terminología de C++. Como Java no soporta los apuntadores, el autor prefiere el término "this reference" (esta referencia).

La palabra clave `this` se puede usar explícitamente cuando se quiere poner en claro que un método accesa sus propias variables y/o métodos. Por ejemplo, se podría reescribir así el método `area()`:

```
public double area() { return 3.14159 * this.r * this.r; }
```

En un método así de sencillo, no es necesario ser explícito. Sin embargo, en casos más complicados, parece que usar un `this` explícito incrementa la claridad del código, aunque no sea estrictamente necesario.

Una instancia donde la palabra clave `this` *sí* se requiere es cuando un argumento de método o una variable local en un método, tiene el mismo nombre que uno de los campos de la clase. En este caso, usted debe usar `this` para tener acceso al campo; porque si usted utilizara nada más el nombre de campo, al final el argumento o la variable local accesada tendría el mismo nombre. En la siguiente sección se verán ejemplos de esto.

Creación de un objeto

Ahora vuelva a examinar la manera como se ha creado el objeto de círculo:

```
Circle c = new Circle();
```

¿Para qué sirven esos paréntesis? ¡Parece como si llamaran una función! Pues, de hecho, eso es exactamente lo que hacen. Toda clase en Java tiene por lo menos un método constructor —con el mismo nombre que la clase—; cuyo propósito es efectuar toda la inicialización necesaria para el nuevo objeto. Como no hubo de definirse una de las clases `Círculo`, Java da un constructor por omisión que ni toma argumentos ni efectúa una inicialización especial.

Ésta es la manera en que funciona: la palabra clave `new` crea una nueva instancia dinámica de la clase; esto es, asigna el objeto nuevo. Posteriormente, se llama al método constructor, se pasa implícitamente el objeto nuevo (una referencia `this`, como se vio con anterioridad) y se pasan explícitamente los argumentos especificados entre paréntesis.

Cómo se define un constructor

Como existe cierta inicialización obvia que podría hacerse para los objetos de círculo, será mejor definir un constructor. En el ejemplo 3-2, se muestra un constructor que permite especificar los valores iniciales para el centro y el radio del nuevo objeto `Círculo`. El ejemplo también presenta un uso de la palabra clave `this`, como se describió en la sección previa.

Ejemplo 3-2: Un constructor para la clase `Círculo`

```
public class Círculo {
    public double x, y, r; // El centro y el radio del círculo

    // El método constructor
    public Círculo (double x, double y, double r)
    {
        this.x = x;
        this.y = y;
        this.r = r;
    }

    public double circunferencia () { return 2 * 3.14159 * r; }
    public double area () { return 3.14159 * r*r; }
}
```

Con el constructor antiguo y por omisión, se tenía que escribir un código como éste:

```
Circle c = new Circle();
c.x = 1.414;
c.y = -1.0;
c.r = .25;
```

Con este nuevo constructor, la inicialización se vuelve parte del paso para crear el objeto:

```
Circle c = new Circle(1.414, -1.0, .25);
```

Existen dos puntos importantes respecto a nombrar y declarar constructores:

- El nombre del constructor es siempre el mismo que el de la clase.
- El tipo de retorno es implícitamente una instancia de la clase. En las declaraciones de constructor, no se especifica un tipo de retorno ni se emplea la palabra clave `void`. El objeto `this` se regresa implícitamente; un constructor no debe usar una declaración `return` para regresar un valor.

Constructores múltiples

En ocasiones, usted querrá inicializar un objeto de distintas maneras, dependiendo de lo que resulte más conveniente en una circunstancia particular. Por ejemplo, si usted desea inicializar el radio de un círculo sin inicializar el centro, inicializar un círculo para que tenga el mismo centro y radio que otro círculo, o inicializar todos los campos con valores por omisión, no hay problema, pues una clase puede tener el número de métodos que sea necesario. En el ejemplo 3-3, se muestra cómo.

Ejemplo 3-3: Varios constructores de Círculo

```
public class Círculo {
    public double x, y, r;

    public Círculo(double x, double y, double r) {
        this.x = x; this.y = y; this.r = r;
    }

    public Círculo(double r) { x = 0.0; y = 0.0; this.r = r; }
    public Círculo(Círculo c) { x = c.x; y = c.y; r = c.r; }
    public Círculo() { x = 0.0; y = 0.0; r = 1.0; }

    public double circunferencia () { return 2 * 3.14159 * r; }
    public double area () { return 3.14159 * r * r; }
}
```

Sobrecarga de método

Lo sorprendente de este ejemplo (quizá no tanto, si usted es un programador en C++) es que todos los métodos de constructor tienen el mismo nombre! ¿Cómo los distingue el compilador? De la misma manera como usted lo hace. Para que usted pueda distinguirlos, observe que los cuatro métodos toman diferentes argumentos y son útiles en distintas circunstancias. En Java, un método se distingue mediante su nombre, el número, el tipo y la posición de sus argumentos. Esto no sólo se aplica a los métodos de constructor, ya que dos métodos no son los mismos, a menos que tengan el mismo nombre y el mismo número de argumentos del mismo tipo, que se pasen en la misma posición en la lista de argumento. Cuando usted llama un método y hay más de uno con igual nombre, el compilador automáticamente selecciona uno que concuerde con los tipos de datos de los argumentos que usted pase.

El hecho de definir métodos con el mismo nombre y distintos tipos de argumento se llama *sobrecarga de método*. Puede resultar una técnica conveniente, siempre y cuando usted sólo dé el mismo nombre a métodos que efectúen funciones similares sobre formas ligeramente distintas de datos de entrada. Los métodos sobrecargados pueden tener distintos tipos de retorno, pero nada más si tienen diferentes argumentos. No confunda la sobrecarga de método con la *invalidación de método* (esta última se describirá más adelante).

Otra vez this

La palabra clave `this` se usa de modo especializado cuando una clase tiene varios constructores; es decir, se puede usar a partir de un constructor para invocar uno de los otros constructores de la misma clase. Así, usted podría reescribir los constructores adicionales del ejemplo 3-3, en términos del primero de ellos:

```
public Circulo(double x, double y, double r) {  
    this.x = x; this.y = y; this.r = r;  
}  
public Circulo(double r) { this(0.0, 0.0, r); }  
public Circulo(Circulo c) { this(c.x, c.y, c.r); }  
public Circulo() { this(0.0, 0.0, 1.0); }
```

Aquí, `this()` hace referencia a cualquier constructor de la clase que tome el tipo especificado de argumentos. Pero, claro, éste sería un ejemplo más convincente si el primer constructor que se invoca hiciera una cantidad más significativa de inicialización, como ocurriría, por ejemplo, si se estuviera escribiendo una clase más complicada.

Existe una restricción muy importante en esta sintaxis de `this`: sólo puede aparecer como la primera declaración en un constructor. Y claro, puede ir seguida de cualquier inicialización adicional que necesite hacer una versión particular del constructor. La razón de esta restricción involucra la invocación automática de métodos de constructor de superclase, que se explorará posteriormente en este capítulo.

Variables de clase

En la definición de la clase `Circulo`, se declararon tres variables de "instancia": `x`, `y` y `r`. Cada instancia de la clase (cada círculo) tiene su propia copia de estas tres variables, las cuales son como los campos de una `struct` en C; es decir, cada instancia de `struct` tiene una copia de los campos. Sin embargo, en ocasiones, se requiere una variable de la cual sólo hay una copia (algo así como una variable global en C).

El problema es que Java no permite variables globales (en realidad, para los conocedores ésta es una característica!). Toda variable en Java se debe declarar dentro de una clase, así que Java usa la palabra clave `static` para indicar que una variable particular es una *variable de clase*, en vez de una *variable de instancia*. Esto es, sólo hay una copia de la variable, asociada con la clase, en vez de muchas copias de la variable asociadas con cada instancia de la clase. La copia única de la variable existe sin importar el número de instancias de la clase que se creen; existe y puede usarse, aunque nunca se cree realmente un caso de la clase.

Este tipo de variable, declarada con la palabra clave `static`, a menudo se denomina *variable estática*. El autor prefiere (y recomienda) el nombre de "variable de clase", ya que se distingue fácilmente de su contraria, la "variable de instancia", porque en este libro se usan ambos términos.

Un ejemplo

Como ejemplo para el caso, suponga que mientras usted desarrollaba la clase `Circulo`, pensó en probarla y determinar con qué frecuencia se usaba. Una manera de hacerlo sería contar el número de casos de objetos `Circulo` creados. Para este fin, obviamente se necesita una variable asociada con la clase y no con una instancia en particular. En el ejemplo 3-4, se muestra cómo hacer esto: se declara una variable `static` que se incrementa cada vez que se crea un objeto `Circulo`.

Ejemplo 3-4: Ejemplo de variable estática

```
public class Circulo {
    static int num_circulos = 0; // variable de clase: cuantos circulos
                                // son creados
    public double x, y, r;       // variables de instancia: el centro
                                // y el radio

    public Circulo(double x, double y, double r) {
        this.x = x; this.y = y; this.r = r;
        num_circulos++;
    }

    public Circulo(double r) { this(0.0, 0.0, r); }
    public Circulo(Circulo c) { this(c.x, c.y, c.r); }
    public Circulo() { this(0.0, 0.0, 1.0); }

    public double circunferencia () { return 2 * 3.14159 * r; }
    public double area () { return 3.14159 * r*r; }
}
```

Cómo tener acceso a variables de clase

Ahora que usted lleva el registro del número de objetos `Circulo` que se crean, ¿cómo puede tener acceso a esta información? A través de la clase y no de la instancia, porque las variables `static` están asociadas con la clase y no con una instancia. Por ello, usted podría escribir:*

```
System.out.println("Numero de circulos creados: " + Circulo.num_circulos);
```

Observe que en la definición del método constructor del ejemplo 3-4, sólo se usó `num_circulos` en vez de `Circulo.num_circulos`; pero aunque se permite hacer esto dentro de la definición de clase de `Circulo`, en cualquier otro sitio debe usarse también el nombre de clase.

¿Variables globales?

Se dijo antes que Java no respalda las variables globales. Sin embargo, en cierto sentido, `Circulo.num_circulos` se comporta como una de ellas. En lo que realmente difiere de una variable global de C, es en que no hay posibilidad de conflictos por nombre. Si se usara alguna otra clase con una variable de clase denominada `num_circulos`, no habría una “colisión” entre estas dos variables “globales”, ya que se haría referencia a ambas con sus nombres de clase. Como cada variable de clase debe ser parte de una clase y uno debe referirse a ella con su nombre de clase, cada variable tiene un nombre único. Por otra parte, cada clase tiene un nombre único porque, como se vio en el capítulo 2, es parte de un paquete con un nombre único.

* Recuerde que `System.out.println()` imprime una línea de texto y que el operador de concatenación de cadena, `+`, convierte los tipos de no cadena en cadenas, según sea necesario.