

sexta
edición

JAVA

PARA

Estudiantes



PEARSON

DOUGLAS BELL & MIKE PARR

Java

para estudiantes

Sexta edición

DOUGLAS BELL
MIKE PARR

Traducción

Alfonso Vidal Romero Elizondo

Ingeniero en Computación

Instituto Tecnológico y de Estudios Superiores de Monterrey

Revisión Técnica

Ing. Jakeline Marcos Abed

Departamento de Ciencias Computacionales

División de Mecatrónica y Tecnologías de Información

Tecnológico de Monterrey, Campus Monterrey

Prentice Hall

México • Argentina • Brasil • Colombia • Costa Rica • Chile • Ecuador
España • Guatemala • Panamá • Perú • Puerto Rico • Uruguay • Venezuela

Datos de catalogación bibliográfica

BELL, DOUGLAS y PARR, MIKE

Java para estudiantes. 6a. edición

PEARSON EDUCACIÓN, México, 2011

ISBN: 978-607-32-0557-3

Área: Computación

Formato: 18.5 × 23.5 cm

Páginas: 552

Authorized translation from the English language edition, entitled *JAVA FOR STUDENTS 06 Edition*, by *Douglas Bell & Mike Parr* published by Pearson Education Limited, United Kingdom © 2010. All rights reserved.
ISBN 027373122X

Traducción autorizada de la edición en idioma inglés, titulada: *JAVA FOR STUDENTS 6^a Edición*, por *Douglas Bell y Mike Parr* publicada por Pearson Education Limited, United Kingdom, Copyright © 2010. Todos los derechos reservados.

Esta edición en español es la única autorizada.

Edición en español

Editor: Luis Miguel Cruz Castillo
e-mail: luis.cruz@pearsoned.com
Editor de desarrollo: Bernardino Gutiérrez Hernández
Supervisor de producción: José D. Hernández Garduño

SEXTA EDICIÓN, 2011

D.R. © 2011 por Pearson Educación de México, S.A. de C.V.
Atlacomulco 500-5o. piso
Col. Industrial Atoto
53519, Naucalpan de Juárez, Estado de México

Cámara Nacional de la Industria Editorial Mexicana. Reg. núm. 1031.

Prentice Hall Europe es una marca registrada de Pearson Educación de México, S.A. de C.V.

Reservados todos los derechos. Ni la totalidad ni parte de esta publicación pueden reproducirse, registrarse o transmitirse, por un sistema de recuperación de información, en ninguna forma ni por ningún medio, sea electrónico, mecánico, fotoquímico, magnético o electroóptico, por fotocopia, grabación o cualquier otro, sin permiso previo por escrito del editor.

El préstamo, alquiler o cualquier otra forma de cesión de uso de este ejemplar requerirá también la autorización del editor o de sus representantes.

Prentice Hall
es una marca de

ISBN VERSIÓN IMPRESA: 978-607-32-0557-3

PRIMERA IMPRESIÓN

Impreso en México. *Printed in Mexico.*

1 2 3 4 5 6 7 8 9 0 - 13 12 11 10



Resumen de contenido

Introducción	xvii
Visita guiada	xxii
1 Antecedentes sobre Java	1
2 Los primeros programas	8
3 Uso de métodos gráficos	22
4 Variables y cálculos	35
5 Métodos y parámetros	60
6 Cómo usar objetos	88
7 Selección	115
8 Repetición	152
9 Cómo escribir clases	171
10 Herencia	194
11 Cálculos	210
12 Objetos de tipo <code>ArrayLists</code>	228
13 Arreglos	242
14 Matrices	265
15 Manipulación de cadenas de texto	278
16 Excepciones	301
17 Archivos y aplicaciones de consola	318
18 Diseño orientado a objetos	348
19 Estilo de los programas	369
20 El proceso de prueba	383

21 El proceso de depuración	397
22 Hilos	406
23 Interfaces	416
24 Programación en gran escala: paquetes	426
25 Polimorfismo	432
26 Java en contexto	441
Apéndices	454
Índice	522

Contenido

Introducción	xvii
Visita guiada	xxii
1 Antecedentes sobre Java	1
La historia de Java	1
Las características principales de Java	2
¿Qué es un programa?	3
Fundamentos de programación	5
Errores comunes de programación	5
Resumen	6
Ejercicios	6
Respuestas a las prácticas de autoevaluación	7
2 Los primeros programas	8
Introducción	8
Entornos de desarrollo integrados	9
Archivos y carpetas	9
Creación de un programa de Java	10
Las bibliotecas	13
Desmitificación del programa	14
Objetos, métodos: una introducción	15
Clases: una analogía	16
Uso de un campo de texto	17
Fundamentos de programación	19
Errores comunes de programación	19
Secretos de codificación	20

Nuevos elementos del lenguaje	20
Resumen	20
Ejercicios	21
Respuestas a las prácticas de autoevaluación	21
3 Uso de métodos gráficos	22
Introducción	22
Eventos	22
El evento de clic de botón	24
El sistema de coordenadas de gráficos	25
Explicación del programa	25
Métodos para dibujar	27
Dibujos a color	28
Creación de un nuevo programa	28
El concepto de secuencia	29
Cómo agregar significado mediante comentarios	31
Fundamentos de programación	31
Errores comunes de programación	32
Secretos de codificación	32
Nuevos elementos del lenguaje	32
Resumen	32
Ejercicios	32
Respuestas a las prácticas de autoevaluación	33
4 Variables y cálculos	35
Introducción	35
La naturaleza de <code>int</code>	36
La naturaleza de <code>double</code>	36
Declaración de variables	37
La instrucción de asignación	41
Cálculos y operadores	41
Los operadores aritméticos	42
El operador %	45
Unión de cadenas con el operador +	46
Conversiones entre cadenas y números	47
Cuadros de mensaje y de entrada	49
Aplicación de formato al texto en cuadros de diálogo mediante \n	51
Conversiones entre números	52
Constantes: uso de final	53
El papel que desempeñan las expresiones	54
Fundamentos de programación	55
Errores comunes de programación	55
Secretos de codificación	56
Nuevos elementos del lenguaje	56
Resumen	57

Ejercicios	57
Respuestas a las prácticas de autoevaluación	59
5 Métodos y parámetros	60
Introducción	60
Cómo escribir sus propios métodos	61
Nuestro primer método	62
Cómo llamar a un método	64
Cómo pasar parámetros	64
Parámetros formales y actuales	66
Un método para dibujar triángulos	67
Variables locales	70
Conflictos de nombres	71
Métodos para manejar eventos y <code>main</code>	72
La instrucción <code>return</code> y los resultados	73
Cómo construir métodos a partir de otros métodos: <code>dibujarCasa</code>	76
Cómo construir métodos a partir de otros métodos: <code>áreaCasa</code>	78
La palabra clave <code>this</code> y los objetos	79
Sobrecarga de métodos	80
Fundamentos de programación	81
Errores comunes de programación	82
Secretos de codificación	82
Nuevos elementos del lenguaje	83
Resumen	83
Ejercicios	83
Respuestas a las prácticas de autoevaluación	86
6 Cómo usar objetos	88
Introducción	88
Variables de instancia	89
Instanciación: uso de constructores con <code>new</code>	92
La clase <code>Random</code>	92
El método <code>main</code> y <code>new</code>	97
El kit de herramientas Swing	98
Eventos	98
Creación de un objeto <code>JButton</code>	99
Lineamientos para usar objetos	101
La clase <code>JLabel</code>	101
La clase <code>JTextField</code>	103
La clase <code>JPanel</code>	104
La clase <code>Timer</code>	104
La clase <code>JSlider</code>	106
La clase <code>ImageIcon</code> – cómo mover una imagen	109
Fundamentos de programación	111
Errores comunes de programación	112

Secretos de codificación	112
Nuevos elementos del lenguaje	112
Resumen	112
Ejercicios	112
Respuestas a las prácticas de autoevaluación	114
7 Selección	115
Introducción	115
La instrucción <code>if</code>	116
<code>if...else</code>	118
Operadores de comparación	121
Múltiples eventos	129
And, or, not	131
Instrucciones <code>if</code> anidadas	134
La instrucción <code>switch</code>	136
Variables booleanas	139
Comparación de cadenas	143
Fundamentos de programación	143
Errores comunes de programación	143
Secretos de codificación	145
Nuevos elementos del lenguaje	146
Resumen	146
Ejercicios	147
Respuestas a las prácticas de autoevaluación	149
8 Repetición	152
Introducción	152
<code>while</code>	153
<code>for</code>	158
And, or, not	159
<code>do...while</code>	161
Ciclos anidados	163
Cómo combinar las estructuras de control	164
Fundamentos de programación	165
Errores comunes de programación	165
Secretos de codificación	166
Nuevos elementos del lenguaje	166
Resumen	167
Ejercicios	167
Respuestas a las prácticas de autoevaluación	169
9 Cómo escribir clases	171
Introducción	171
Cómo diseñar una clase	172

Clases y archivos	175
Variables <code>private</code>	177
Métodos <code>public</code>	177
Los métodos <code>get</code> y <code>set</code>	179
Constructores	180
Varios constructores	181
Métodos <code>private</code>	182
Reglas de alcance	184
Operaciones sobre los objetos	185
Destrucción de objetos	186
Métodos <code>static</code>	186
Variables <code>static</code>	187
Fundamentos de programación	188
Errores comunes de programación	189
Secretos de codificación	190
Nuevos elementos del lenguaje	190
Resumen	191
Ejercicios	191
Respuestas a las prácticas de autoevaluación	193
10 Herencia	194
Introducción	194
Cómo usar la herencia	195
<code>protected</code>	196
Reglas de alcance	197
Elementos adicionales	197
Redefinición (o sobreescritura)	198
Diagramas de clases	198
La herencia en acción	199
<code>super</code>	200
Constructores	200
<code>final</code>	203
Clases abstractas	204
Principios de programación	205
Errores comunes de programación	206
Nuevos elementos del lenguaje	207
Resumen	207
Ejercicios	208
Respuestas a las prácticas de autoevaluación	209
11 Cálculos	210
Introducción	210
Funciones y constantes de la biblioteca matemática	211
Cómo aplicar formato a los números	211
Ejemplo práctico: dinero	214

Ejemplo práctico – iteración	217
Gráficos	218
Excepciones	222
Fundamentos de programación	223
Errores comunes de programación	223
Resumen	223
Ejercicios	224
Respuesta a la práctica de autoevaluación	227
12 Objetos de tipo ArrayLists	228
Introducción	228
Creación de un objeto ArrayList y los tipos genéricos	229
Cómo agregar elementos a una lista	229
La longitud de una lista	230
Índices	231
Cómo mostrar un objeto ArrayList	231
La instrucción for mejorada	232
Cómo utilizar valores de índice	233
Cómo eliminar elementos de un objeto ArrayList	234
Cómo insertar elementos dentro de un objeto ArrayList	235
Búsquedas rápidas (lookup)	235
Operaciones aritméticas en un objeto ArrayList	236
Búsquedas detalladas (search)	238
Fundamentos de programación	239
Errores comunes de programación	240
Nuevos elementos del lenguaje	240
Resumen	240
Ejercicios	241
Respuestas a las prácticas de autoevaluación	241
13 Arreglos	242
Introducción	242
Cómo crear un arreglo	244
Índices	245
La longitud de un arreglo	247
Cómo pasar arreglos como parámetros	247
La instrucción for mejorada	248
Uso de constantes con los arreglos	249
Cómo inicializar un arreglo	250
Un programa de ejemplo	251
Búsqueda rápida (lookup)	253
Búsqueda detallada (search)	254
Arreglos de objetos	256
Fundamentos de programación	257
Errores comunes de programación	258

Secretos de codificación	259
Resumen	259
Ejercicios	260
Respuestas a las prácticas de autoevaluación	263
14 Matrices	265
Introducción	265
Cómo declarar una matriz	266
Índices	267
El tamaño de una matriz	268
Paso de matrices como parámetros	269
Uso de constantes con matrices	269
Cómo inicializar una matriz	270
Un programa de ejemplo	271
Fundamentos de programación	272
Errores comunes de programación	273
Resumen	273
Ejercicios	274
Respuestas a las prácticas de autoevaluación	277
15 Manipulación de cadenas de texto	278
Introducción	278
Uso de cadenas de texto — un recordatorio	279
Los caracteres dentro de cadenas de texto	280
Una observación sobre el tipo <code>char</code>	280
La clase <code>String</code>	281
Los métodos de la clase <code>String</code>	281
Comparación de cadenas	283
Corrección de cadenas	285
Análisis de cadenas	286
Conversiones de cadenas	289
Parámetros de cadena	291
Un ejemplo de procesamiento de cadenas	291
Ejemplo práctico: Frasier	292
Fundamentos de programación	296
Errores comunes de programación	297
Secretos de codificación	297
Nuevos elementos del lenguaje	297
Resumen	298
Ejercicios	298
Respuesta a la práctica de autoevaluación	300
16 Excepciones	301
Introducción	301
Excepciones y objetos	303

Cuándo usar excepciones	304
La jerga de las excepciones	304
Un ejemplo con <code>try-catch</code>	304
<code>try</code> y los alcances	307
La búsqueda de un bloque <code>catch</code>	308
Lanzamiento de excepciones: una introducción	309
Clases de excepciones	310
La compilación y las excepciones verificadas	310
Cómo atrapar excepciones – los casos comunes	312
Uso de la estructura de las clases de excepciones	314
Fundamentos de programación	314
Errores comunes de programación	315
Secretos de codificación	315
Nuevos elementos del lenguaje	315
Resumen	316
Ejercicios	316
Respuestas a las prácticas de autoevaluación	317
17 Archivos y aplicaciones de consola	318
Introducción	318
Acceso a los archivos: ¿flujo o aleatorio?	319
Aspectos esenciales de los flujos	319
Las clases de E/S de Java	320
Las clases <code>BufferedReader</code> y <code>PrintWriter</code>	320
Salida de archivos	321
Entrada de archivos	324
Operaciones de búsqueda con archivos	327
La clase <code>File</code>	331
La clase <code>JFileChooser</code>	333
E/S de consola	336
La clase <code>System</code>	336
Uso de <code>JOptionPane</code>	338
Un ejemplo de consola: <code>Buscador</code>	338
Cómo leer desde un sitio remoto	340
Argumentos de la línea de comandos	342
Fundamentos de programación	344
Errores comunes de programación	344
Secretos de codificación	344
Nuevos elementos del lenguaje	344
Resumen	345
Ejercicios	346
Respuestas a las prácticas de autoevaluación	347
18 Diseño orientado a objetos	348
Introducción	348
El problema del diseño	349

Cómo identificar los objetos y métodos	349
Ejemplo práctico de diseño	354
En búsqueda de la reutilización	360
¿Composición o herencia?	361
Lineamientos para el diseño de clases	365
Resumen	366
Ejercicios	367
Respuestas a las prácticas de autoevaluación	368
19 Estilo de los programas	369
Introducción	369
Distribución del programa	370
Nombres	370
Clases	371
Comentarios	372
Javadoc	373
Constantes	373
Métodos	374
Instrucciones <code>if</code> anidadas	375
Ciclos anidados	378
Condiciones complejas	379
Documentación	381
Consistencia	381
Errores comunes de programación	382
Resumen	382
Ejercicios	382
20 El proceso de prueba	383
Introducción	383
Especificaciones de los programas	384
Prueba exhaustiva	385
Prueba de la caja negra (funcional)	385
Prueba de la caja blanca (estructural)	388
Inspecciones y recorridos	390
Paso a paso por instrucciones	391
Desarrollo incremental	391
Principios de programación	392
Resumen	392
Ejercicios	393
Respuestas a las prácticas de autoevaluación	394
21 El proceso de depuración	397
Introducción	397
Cómo depurar sin un depurador	399
Cómo usar un depurador	400
Errores comunes – errores de compilación	401

Errores comunes – errores en tiempo de ejecución	402
Errores comunes – errores lógicos	403
Errores comunes – malentender el lenguaje	403
Resumen	405
Respuesta a la práctica de autoevaluación	405
22 Hilos	406
Introducción	406
Hilos	407
Cómo iniciar un hilo	411
Muerte de un hilo	412
join	412
El estado de un hilo	412
Planificación de procesos, prioridades de hilos y yield	413
Principios de programación	414
Resumen	414
Ejercicios	415
Respuestas a las prácticas de autoevaluación	415
23 Interfaces	416
Introducción	416
Interfaces para el diseño	416
Interfaces e interoperabilidad	419
Las interfaces y la biblioteca de Java	420
Interfaces múltiples	421
Comparación entre interfaces y clases abstractas	423
Fundamentos de programación	423
Errores comunes de programación	423
Secretos de codificación	424
Nuevos elementos del lenguaje	424
Resumen	424
Ejercicios	424
Respuestas a las prácticas de autoevaluación	425
24 Programación en gran escala: paquetes	426
Introducción	426
Uso de clases y la instrucción import	426
Creación de paquetes mediante la instrucción package	427
Paquetes, archivos y carpetas	428
Reglas de alcance	429
Los paquetes de la biblioteca de Java	429
Errores comunes de programación	430
Nuevos elementos del lenguaje	430
Resumen	430
Ejercicio	430
Respuestas a las prácticas de autoevaluación	431

25 Polimorfismo	432
Introducción	432
El polimorfismo en acción	433
Fundamentos de programación	437
Errores comunes de programación	438
Nuevos elementos del lenguaje	438
Resumen	439
Ejercicios	439
26 Java en contexto	441
Introducción	441
Simple	442
Orientado a objetos	442
Independencia de la plataforma (portabilidad)	442
Rendimiento	443
Seguridad	444
Código fuente abierto	446
Las versiones de Java	446
Herramientas de Java	447
Bibliotecas de Java	447
Java beans	447
Bases de datos – JDBC	448
Java e Internet	449
Java y World Wide Web	450
La oposición: plataforma .NET de Microsoft	451
JavaScript	452
Conclusión	453
Resumen	453
Ejercicios	453
Apéndices	
A Bibliotecas de Java	454
B El Abstract Window Toolkit	496
C Applets	500
D Glosario	504
E Reglas para los nombres	506
F Palabras clave	507
G Reglas de alcance (visibilidad)	508
H Bibliografía	511
I Instalación y uso de Java	513
Índice	522

Introducción

● Lo que este libro le dirá

En este libro le explicaremos cómo escribir programas de Java que se ejecuten como aplicaciones independientes o como applets (parte de una página Web).

● Este libro es para principiantes

Si nunca ha programado —si es todo un principiante—, este libro es para usted. No necesita tener conocimientos previos sobre programación ya que aquí se explican los conceptos desde cero en un estilo simple y directo para una máxima claridad. El libro está dirigido principalmente a estudiantes universitarios de primer nivel, pero también es adecuado para principiantes que quieren aprender por su cuenta.

● ¿Por qué Java?

Java es tal vez uno de los mejores lenguajes para aprender y usar debido a las siguientes características.

Java es pequeño y hermoso

Los diseñadores de Java omitieron a propósito todas las características superfljas de los lenguajes de programación; recortaron el diseño al máximo. El resultado es un lenguaje que cuenta con todas las herramientas necesarias, combinadas en forma elegante y lógica. El diseño está optimizado y perfeccionado y es fácil de aprender, pero muy poderoso.

Java está orientado a objetos

Los lenguajes orientados a objetos son la metodología más reciente y exitosa, y su programación es la más popular. Java fue concebido totalmente orientado a objetos; no es un lenguaje al que se le haya agregado este enfoque como una idea de último momento.

Java cuenta con soporte para Internet

Una importante motivación para Java es permitir que las personas desarrollen programas que utilicen Internet y World Wide Web. Los applets de Java se pueden invocar con facilidad desde los navegadores Web para proveer herramientas valiosas y espectaculares. Además, los programas de Java se pueden transmitir fácilmente por Internet para ejecutarse en cualquier computadora.

Java es de propósito general

Java es un verdadero lenguaje de propósito general. Cualquier cosa que puedan hacer C++, Visual Basic, y otros lenguajes, también lo puede hacer Java.

Java es independiente de la plataforma

Los programas de Java se pueden ejecutar en casi cualquier computadora y teléfono celular, y en casi todos los sistemas operativos... ¡sin que haya necesidad de modificarlos! Intente hacer eso con cualquier otro lenguaje de programación (¡es casi imposible!). Esto se sintetiza mediante el eslogan “escribalo una vez – ejecútelo en cualquier parte”.

Java cuenta con bibliotecas

Como Java es un lenguaje pequeño, la mayor parte de su funcionalidad se provee mediante piezas de programas contenidas en bibliotecas. Hay una gran cantidad de software de biblioteca disponible para crear gráficos, acceder a Internet, proveer interfaces gráficas de usuario (GUIs) y muchas otras cosas más.

● Requerimientos

Para aprender a programar necesita una computadora y ciertos paquetes de software. Un sistema común es una PC (computadora personal) con el Kit de Desarrollo de Software de Java (JDK). Este kit también está disponible para sistemas Unix, GNU/Linux y Apple, y le permite preparar y ejecutar programas de Java. También existen otros entornos de desarrollo más accesibles. Consulte el capítulo 2.

● Los ejercicios son buenos para usted

Aunque usted leyera este libro varias veces hasta que pudiera recitarlo al revés, no por ello podría escribir programas. El trabajo práctico relacionado con la escritura de programas es vital para obtener fluidez y confianza en la programación.

Al final de cada capítulo encontrará una serie de ejercicios para practicar. Le invitamos a que realice algunos de ellos para que mejore su habilidad.

También hemos incluido pequeñas prácticas de autoevaluación a lo largo del libro con sus respuestas al final de cada capítulo para que se cerciore de haber entendido las cosas en forma apropiada.

● Qué incluye este libro

Este libro explica los conceptos fundamentales de la programación:

- Variables.
- Asignación.
- Entrada y salida.
- Cálculos.
- Programación de gráficos y ventanas.
- Selección mediante `if`.
- Repeticiones mediante `while`.

Además, explica cómo utilizar números enteros, números de punto flotante y cadenas de caracteres, y describe los arreglos. Todos estos temas son fundamentales, sin importar el tipo de programación que usted lleve a cabo.

También explica con detalle los aspectos orientados a objetos de la programación:

- El uso de las clases de la biblioteca.
- La escritura de clases.
- El uso de objetos.
- El uso de métodos.

Además veremos algunos de los aspectos más sofisticados de la programación orientada a objetos:

- Herencia.
- Polimorfismo.
- Interfaces.

● Qué no incluye este libro

Este libro se limita a los aspectos esenciales de Java. Por ende, no explica todos los aspectos específicos relacionados con este lenguaje. De esta manera el lector no tiene que lidiar con los detalles innecesarios para concentrarse en dominar Java y la programación en general.

● ¿Aplicaciones o applets?

Hay dos tipos diferentes de programas de Java:

- Un programa independiente y definido (al cual se le conoce como aplicación).
- Un programa que se invoca desde un navegador Web (a este se le conoce como applet).

En este libro nos concentraremos en las aplicaciones, ya que consideramos que constituyen la principal forma en la que se utiliza Java (en el apéndice C le explicaremos cómo ejecutar applets).

● ¿Gráficos o texto?

A lo largo del texto hacemos énfasis en programas que utilizan imágenes gráficas en vez de entrada y salida de texto. Creemos que son más divertidos, interesantes y que muestran con claridad todos los principios importantes de la programación. No hemos ignorado los programas que reciben y producen texto; incluimos algunos de ellos, sólo que en segundo término.

● Interfaces gráficas de usuario (GUI)

Los programas que presentaremos utilizan muchas de las características de una GUI, como ventanas, botones y barras de desplazamiento; también utilizan el ratón en muchas formas distintas.

● ¿AWT o Swing?

Existen dos mecanismos de Java para crear y utilizar GUI: AWT y Swing. El conjunto de componentes de interfaz de usuario de Swing es más completo y poderoso que el conjunto del AWT. En este libro adoptaremos la metodología de Swing, ya que es la más utilizada en la actualidad.

● La secuencia del material

La programación implica muchos conceptos desafiantes; uno de los problemas al escribir un libro sobre programación es decidir cómo y cuándo introducirlos. Presentaremos los conceptos simples en los primeros capítulos y los más sofisticados al final. Utilizaremos los objetos desde el principio y después veremos cómo escribir nuevos objetos. Nuestra metodología es empezar con conceptos como variables y asignaciones, para después introducir las selecciones y los ciclos, y posteriormente pasar a los objetos y las clases (las características orientadas a objetos). También quisimos asegurarnos de que la diversión fuera el elemento más importante de la programación, por lo que utilizamos gráficos desde el inicio.

● Detalle a detalle

En este libro presentamos los conceptos cuidadosamente, uno por uno, en vez de verlos todos juntos. Por ejemplo, encontrará que hay un capítulo entero sobre cómo escribir métodos.

● Aplicaciones de computadora

Las computadoras se utilizan en muchas aplicaciones; en este libro utilizamos ejemplos de las siguientes áreas:

- Procesamiento de información.
- Juegos.
- Cálculos científicos.

Usted puede optar por concentrarse en las áreas de aplicación que sean de su interés e invertir menos tiempo en las demás.

● Distintos tipos de programación

Hay muchos tipos de programación; algunos ejemplos son: programación por procedimientos, lógica, funcional, de hojas electrónicas de cálculo, visual y orientada a objetos. En este libro hablaremos sobre el tipo dominante: la programación orientada a objetos, POO, tal como se practica en lenguajes como Visual Basic, C++, C#, Eiffel y Smalltalk.

● ¿Cuál versión de Java?

Utilizamos Java 6.

● Diviértase

La programación es un proceso creativo e interesante, especialmente con Java. Así que ¡diviértase!

● Visite nuestro sitio Web

Todos los programas presentados en este libro están disponibles en nuestro sitio Web, el cual puede visitar en: <http://www.pearsoneducacion.net/bell/>

● Cambios en esta edición

Si leyó ediciones anteriores de este libro, tal vez quiera saber qué diferencias tiene ésta.

La versión más reciente de Java es la 6, y este libro se basa en esa versión. No hay cambios en cuanto al lenguaje o a las clases de biblioteca que utilizamos, pero todos los programas de este libro funcionan con la versión 6.

Los cambios principales en esta edición son:

- El capítulo 2, “Los primeros programas” y el apéndice I. Mejoramos la explicación para incluir material sobre los entornos de desarrollo integrados (IDE).
- El CD. En una época de banda ancha, eliminamos el CD. Todo el contenido (y más) lo encontrará en el sitio Web.
- El capítulo 26, sobre el papel que desempeña Java en el mundo, se actualizó minuciosamente.
- Hay mejoras a lo largo de todo el libro para incrementar su comprensión.

Esperamos le gusten los cambios.

Visita guiada

76 | Capítulo 5 ■ Métodos y parámetros

Ejercicios | 147

PRÁCTICA DE AUTOEVALUACIÓN

- 5.7 El siguiente método se llama `doble` y devuelve el doble del valor de su parámetro `int`:
- ```
private int doble(int n) {
 return 2 * n;
}
```

Dadas las siguientes llamadas al método:

```
int n = 3;
int r = doble(n);
r = doble(n + 1);
r = doble(n + 1);
r = doble(n + 1);
r = doble(doble(n));
r = doble(doble(n - 1));
r = doble(doble(n + 1));
r = doble(doble(doble(n))));
```

Indique el valor devuelto por cada llamada.

### Cómo construir métodos a partir de otros métodos: dibujarCasa

Como ejemplo de métodos que utilizan otros métodos, vamos a crear un método que dibuja una casa sencilla con techo triangular, la cual se muestra en la figura 5.7. La altura del techo es la mitad de la altura de las paredes, y la anchura de las paredes es la misma que la anchura del techo. Vamos a utilizar los siguientes parámetros `int`:

- La posición horizontal del punto superior derecho del techo.
- La posición vertical del punto superior derecho del techo.



Figura 5.7 Una casa cuya anchura es de 100 y la altura del techo es de 50.

Muchas prácticas de autoevaluación distribuidas por todo el libro, además de los ejercicios al final de cada capítulo, permiten al estudiante practicar con los conceptos hasta que los comprenda por completo. Las respuestas a las prácticas de autoevaluación aparecen al final de cada capítulo.

Los nuevos elementos del lenguaje reiteran las nuevas características de sintaxis que se presentan en el capítulo.

350 | Capítulo 18 ■ Diseño orientado a objetos

Resumen | 207

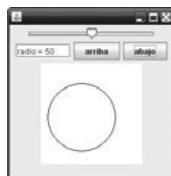


Figura 18.1 El programa del globo.

Por ejemplo, a continuación le mostramos la especificación para el sencillo programa del globo:

Escriba un programa para representar un globo y manipularlo mediante una GUI. El globo debe mostrarse como un círculo en un panel. Utilice botones para cambiar la posición del globo y moverlo una distancia fija hacia arriba o hacia abajo. Use un control deslizable para modificar el radio del globo, el cual se debe mostrar en un campo de texto.

La ventana se muestra en la figura 18.1.

Debenemos buscar verbos y sustantivos en la especificación. En este caso podemos ver los siguientes sustantivos:

`GLOBO`, `panel`, `botón`, `control deslizable`, `campo de texto`, `globo`, `posición`, `distanzia`, `radio`.

La GUI provee la interfaz de usuario para el programa. Consiste en botones, un control deslizable, un campo de texto y un panel. La GUI se representa mediante un objeto que es una instancia de la clase `JFrame`. Los objetos `botón`, `control deslizable`, `campo de texto` y `panel` están disponibles como clases en la biblioteca de Java.

El objeto `GLOBO`:

1. Crea los botones, el control deslizable, el campo de texto y el panel en la pantalla.
2. Maneja los eventos de los clics de ratón en los botones y el control deslizable.
3. Crea cualquier otro objeto que se necesite, como el objeto `globo`.
4. Llama a los métodos del objeto `globo`.

El siguiente objeto importante es el globo, que utiliza información para representar su posición (coordenadas `x` y `y`), la distancia que se mueve y su radio. Una opción sería crear varios objetos completamente distintos para representar estos elementos. Pero es más simple representarlos como variables `int`.

### Nuevos elementos del lenguaje

- `extends`: indica que esta clase hereda de otra clase.
- `protected`: la descripción de una variable o un método que se pueden utilizar dentro de la clase o de cualquier subclase (pero no desde ningún otro lado).
- `abstract`: la descripción de una clase abstracta que no se puede crear, es decir, que no es instanciable, sino que se proporciona sólo para utilizarse en la herencia.
- `abstract`: la descripción de un método que se proporciona sólo como encabezado y debe ser proporcionado por una subclase.
- `super`: el nombre de la superclase de una clase, la clase de la que hereda.
- `final`: describe a un método o una variable que no se puede redefinir.

### Resumen

Extender (heredar) las herramientas de una clase es una buena forma de utilizar las partes existentes de la clase.

Una subclase hereda las herramientas de su superclase inmediata y de todas las superclases por encima de ella en el árbol de herencia.

Una clase sólo tiene una superclase inmediata (sólo puede heredar de una clase). A esto se le conoce como *herencia simple* en la jerga de la POO.

Una clase puede extender las herramientas de una clase existente si proporciona uno o más de los siguientes elementos:

- Métodos adicionales.
- Variables adicionales.
- Métodos que redifinen a *acáñan en vez de* los métodos de la superclase.

Una variable o un método se pueden describir con uno de los siguientes tres tipos de acceso:

- `public`: se puede utilizar desde cualquier clase.
- `private`: se puede utilizar sólo dentro de esta clase.
- `protected`: se puede utilizar sólo dentro de esta clase y de cualquier subclase.

Un diagrama de clases es un árbol que muestra las relaciones de herencia.

Para hacer referencia al nombre de la superclase de una clase se utiliza la palabra `super`.

Una clase abstracta se define con la palabra `abstract`. No se puede instanciar para producir un objeto, ya que `class` es incompleta. Dicha clase provee variables y métodos útiles que las subclases pueden heredar.

En todo el libro se utilizan **programas** con imágenes gráficas (específicamente GUI) en vez de programas de entrada-salida de texto. Esto demuestra el lado creativo y emocionante de la programación, lo cual ayuda a que el estudiante aprenda los conceptos con más rapidez.

Los **resúmenes** ofrecen una síntesis concisa de los conceptos clave cubiertos en cada capítulo. Se enlazan con los objetivos que se listan al principio del capítulo y son una excelente referencia, además de una herramienta de repaso.



Figura 7.11 El anuncio de una tienda.

Al hacer clic en el botón Cerrada:

```
abierta = false;
```

Al hacer clic en el botón Prendido, el valor de `abierta` se evalúa mediante una instrucción `if` y se muestra el anuncio apropiado:

```
if (abierta) {
 campoTexto.setText("Abierta");
} else {
 campoTexto.setText("Cerrada");
}
```

He aquí el programa completo:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class AnuncioTienda extends JFrame implements ActionListener {
 private JButton botonPrendido, botonApagado, botonAbierta, botonCerrada;
 private JTextField campoTexto;
 private boolean prendido = false, abierta = false;
 public static void main(String[] args) {
 ApplicationFrame demo = new ApplicationFrame();
 demo.setSize(300, 200);
 demo.setVisible(true);
 }
}
```

```
private void crear() {
 setDefaultCloseOperation(EXIT_ON_CLOSE);
 Contarles ventana = getContentPane();
 ventana.setLayout(new FlowLayout());
 botonPrendido = new JButton("Prendido");
 botonPrendido.addActionListener(this);
 botonApagado = new JButton("Apagado");
 botonApagado.addActionListener(this);
 ventana.add(botonPrendido);
 ventana.add(botonApagado);
 botonAbierta = new JButton("Abierta");
 campoTexto = new JTextField(5);
 campoTexto.setEditable(false);
 campoTexto.setFont(null, Font.BOLD, 60);
 ventana.add(campoTexto);
 botonAbierta = new JButton("Abierta");
 ventana.add(botonAbierta);
 botonAbierta.addActionListener(this);
 botonCerrada = new JButton("Cerrada");
 ventana.add(botonCerrada);
 botonCerrada.addActionListener(this);
}
```

```
public void actionPerformed(ActionEvent event) {
 Object origen = event.getSource();
 if (origen == botonPrendido) {
 nombrejarBotonPrendido();
 } else if (origen == botonApagado) {
 nombrejarBotonApagado();
 } else if (origen == botonAbierta) {
 nombrejarBotonAbierta();
 } else if (origen == botonCerrada) {
 nombrejarBotonCerrada();
 darsejunto();
 }
}

private void nombrejarBotonPrendido() {
 prendido = true;
}
private void nombrejarBotonApagado() {
 prendido = false;
}
private void nombrejarBotonAbierta() {
 abierta = true;
}
private void nombrejarBotonCerrada() {
 abierta = false;
}
```

**Los errores comunes de programación** hacen énfasis en éstos y en la forma de evitarlos.

Se incluye **código de ejemplo** en el libro; en esta edición se utiliza Swing en todo el código.

### Errores comunes de programación

- El encabezado del método debe incluir los nombres de los tipos. El siguiente código está mal:
 

```
private void métodoName(x) // incorrecto
```

 Debemos utilizar algo como esto:
 

```
private void métodoName(int x) {
```
- La llamada a un método no debe incluir los nombres de los tipos. Por ejemplo, en vez de:
 

```
métodoName(int y); //
```

 debemos marcar:
 

```
métodoName(y);
```
- Al llamar a un método debemos suministrar el número correcto de parámetros y los tipos correctos de estos.
- Siempre debemos consumir un valor devuelto de alguna forma. El siguiente estilo de llamar no consume un valor devuelto:
 

```
unMétodo(a, f); //
```

### Secretos de codificación

- El patrón general para los métodos toma dos formas. En primer lugar, para un método que no devuelve un resultado, la forma de declararlo es:
 

```
private void nombreMétodo(lista de parámetros formales) {
 ...
}
```

 y debemos invocar el método mediante una instrucción, como en el siguiente ejemplo:
 

```
nombreMétodo(lista de parámetros actuales);
```
- Para un método que devuelve un resultado, la forma de declararlo es:
 

```
private tipo nombreMétodo(lista de parámetros actuales) {
 ...
}
```

 Se puede especificar cualquier tipo o clase para el valor devuelto. Podemos invocar el método como parte de una expresión, como en el siguiente ejemplo:
 

```
n = nombreMétodo(a, b);
```

**Los secretos de codificación** identifican la forma correcta de escribir código para reforzar la comprensión del estudiante en cuanto a la sintaxis de Java.



### Applets

#### Introducción

Este libro trata acerca de escribir "aplicaciones". Se ejecutan bajo el control de su sistema operativo y el código de Java: los archivos de clase correspondientes se almacenan en su computadora. Los applets son distintos. El término significa programa pequeño. Los archivos de clase de los applets compilados se cargan en una computadora que actúa como servidor Web, en la misma carpeta en que podría almacenar sus páginas Web. Podemos especificar que una página Web incluya un vínculo a un applet. Cuando un usuario descarga dicha página, se incluirá con ella el código de las clases de Java y el applet se ejecutará en un área de la ventana del navegador Web. He aquí el código:

```
import java.awt.*;
import java.applet.Applet;
import java.awt.event.*;
public class AppletSumarCamposTexto
 extends Applet implements ActionListener {
 private JTextField campoNúmero1, campoNúmero2, camposuma;
 private Label etiquetaSalida;
 private Button botónSumar;
 ...
}
```

500

**Los apéndices** amplían la comprensión del estudiante sobre la programación en Java.



# CAPÍTULO 1



## Antecedentes sobre Java

En este capítulo conoceremos:

- Cómo y por qué surgió Java.
- Las características principales de Java.
- Los conceptos preliminares sobre programación.

### ● La historia de Java

Un programa computacional consiste en una serie de instrucciones que una computadora lleva a cabo. El objetivo de las instrucciones es realizar una tarea; por ejemplo, jugar un juego, enviar un correo electrónico, etc. Las instrucciones se redactan con un estilo particular: se deben escribir conforme a las reglas del lenguaje de programación que elijamos. Hay cientos de lenguajes de programación, pero sólo unos cuantos han dejado huella y se han vuelto muy populares. La historia de los lenguajes de programación es una forma de evolución, y aquí veremos las raíces de Java. Los nombres de los lenguajes anteriores no son importantes, sin embargo, los mencionaremos para que el lector tenga una idea más amplia sobre el tema.

Durante la década de 1960 se creó un lenguaje de programación llamado Algol 60 (el término “Algol” proviene de “algoritmo”—una serie de pasos que se pueden llevar a cabo para resolver un problema—). Este lenguaje fue popular en los círculos académicos, aunque sus conceptos persistieron mucho más tiempo que su uso. En esa época otros lenguajes eran más populares: COBOL para el procesamiento de datos y Fortran para el trabajo científico. En el Reino Unido se creó una versión extendida de Algol 60 (CPL o lenguaje de programación combinado), cuyo nombre se simplificó casi de inmediato a CPL básico o BCPL.

Ahora pasemos a los Laboratorios Bell en Estados Unidos, en donde Dennis Ritchie y otros colaboradores transformaron a BCPL en un lenguaje llamado B, que posteriormente se mejoró para convertirse en C durante la década de 1970. C era enormemente popular. Se utilizó para escribir el sistema operativo Unix, y mucho después Linus Torvalds lo utilizó para escribir una gran parte de Unix —conocida como Linux— para PC.

El siguiente avance fue cuando Bjarne Stroustrup creó C++ durante la década de 1980, también en los Laboratorios Bell. Este lenguaje hizo posible la creación y reutilización de secciones independientes de código, en un estilo conocido como “programación orientada a objetos” (en C se podía utilizar el operador ++ para sumar uno a un elemento —de aquí que C++ sea uno más que C—). C++ es popular aún, pero difícil de usar; se requiere de mucho estudio.

Ahora pasemos a Sun Microsystems en Estados Unidos. A principios de la década de 1990, James Gosling diseñó un nuevo lenguaje conocido como Oak para utilizarlo en productos electrónicos para el consumidor. Este proyecto nunca llegó a realizarse, pero el lenguaje Oak cambió su nombre a Java (en honor al café).

Al mismo tiempo, la popularidad de Internet iba en aumento y una pequeña empresa llamada Netscape había creado un navegador Web.

Después de negociaciones con Microsoft, Netscape acordó proveer soporte para Java en su navegador Web, y el resultado fue que se podían descargar programas de Java junto con las páginas Web. Gracias a ello se pudo ofrecer una herramienta de programación para mejorar las páginas estáticas. Estos programas se conocían como ‘applets’. Netscape decidió permitir que los usuarios descargaran su navegador Web sin costo, lo cual favoreció que Java se diera a conocer más.

## ● Las características principales de Java

Cuando James Gosling diseñó Java, no creó algo de la nada sino que tomó los conceptos existentes y los integró para formar un nuevo lenguaje. He aquí sus características principales:

- Los programas de Java son similares a los programas de C++. Esto significaba que la comunidad de C++ lo tomaría en serio y también que los programadores de C++ podrían obtener resultados con rapidez.
- Java se diseño con Internet en mente. Además de poder crear programas convencionales, también se pueden crear applets que se ejecutan “dentro” de una página Web. Java también cuenta con herramientas para transferir datos a través de Internet de varias formas.
- Los programas de Java son portables: se pueden ejecutar en cualquier tipo de computadora. Para que esto ocurra debe escribirse un ‘sistema en tiempo de ejecución’ o máquina virtual de Java para cada tipo de computadora, y es lo que se ha hecho prácticamente para todos los tipos de computadoras que se utilizan en la actualidad. Java también está disponible para teléfonos celulares o móviles, por lo que, en cierto sentido, el abandonado proyecto Oak finalmente se pudo realizar.
- Los applets de Java son seguros. Los virus de computadora se esparcen con mucha facilidad, por lo que puede ser riesgoso descargar y ejecutar programas a través de Internet. Sin embargo, el diseño de los applets de Java implica que son seguros y que no infectarán su computadora con un virus.

■ Aunque no es una cuestión técnica, las labores de marketing de Sun en relación con Java son dignas de mencionar. Todo el software necesario para crear y ejecutar programas de Java está disponible en forma gratuita, como descarga a través de Internet. Esto ayudó a que Java se hiciera popular rápidamente. Además, el navegador Web de Netscape ofrecía soporte para Java desde sus inicios y Microsoft ofreció herramientas similares en su navegador Web Internet Explorer.

Java tuvo un excelente recibimiento en la industria debido a su portabilidad y características de Internet. También fue bien recibido en la educación, ya que provee herramientas completas orientadas a objetos de una manera más sencilla que C++.

Aunque Java es relativamente nuevo, influyó en el diseño del lenguaje C# (C Sharp) de Microsoft. Desde un punto de vista educativo, al estar familiarizado con Java podrá aprender C# con relativa facilidad.

## ● ¿Qué es un programa?

En esta sección trataremos de dar al lector una idea sobre lo que es un programa. Una forma de comprenderlo es mediante el uso de analogías con recetas, partituras musicales y patrones de bordado. Incluso las instrucciones en una botella de champú para el cabello constituyen un programa simple:

```
mojar el cabello
aplicar champú
dar masaje para incorporar el champú
enjuagar
```

Este programa es una lista de instrucciones para un ser humano, pero demuestra un aspecto importante de un programa de computadora: un programa es una secuencia de instrucciones que se llevan a cabo, empezando en la primera instrucción y avanzando de una instrucción a otra hasta completar la secuencia. Una receta, una partitura musical y un patrón de bordado son similares en cuanto a que constituyen una lista de instrucciones que se realizan en secuencia. En el caso de un patrón de bordado, existen máquinas de bordado que reciben un programa de instrucciones y después lo llevan a cabo (o “ejecutan”). Así es como funciona una computadora: es una máquina que ejecuta de manera automática una secuencia de instrucciones, o un programa (de hecho, si cometemos un error en las instrucciones, es probable que la computadora realice una tarea incorrecta). El conjunto de instrucciones disponibles para que una computadora las lleve a cabo por lo general se compone de:

- Recibir un número como entrada.
- Recibir algunos caracteres como entrada (letras y dígitos).
- Mostrar algunos caracteres como resultado.
- Realizar un cálculo.
- Mostrar un número como resultado.
- Mostrar una imagen gráfica en la pantalla.
- Responder a un botón en la pantalla cuando se hace clic en él mediante el ratón.

La tarea de programar consiste en seleccionar de esta lista aquellas instrucciones que lleven a cabo la tarea requerida. Estas instrucciones se escriben en un lenguaje especializado, conocido como lenguaje de programación. Java es uno de estos lenguajes. Aprender a programar significa aprender sobre las herramientas del lenguaje de programación y cómo combinarlas de manera que realicen lo que usted desea. El ejemplo de las partituras musicales ilustra otro aspecto de los programas. Es común en la música que se repitan algunas secciones; por ejemplo, la sección del coro. La notación musical evita que el compositor tenga que duplicar las partes de la partitura que se repiten, pues proporciona una notación para especificar que se va a repetir una sección de música. Lo mismo se aplica en un programa; a menudo se da el caso de que se tiene que repetir cierta acción: por ejemplo, buscar una palabra dentro de un pasaje de texto, en un programa de procesamiento de texto. La repetición (o iteración) es común en los programas, y Java cuenta con instrucciones especiales para llevar a cabo estas tareas.

Algunas veces las recetas indican algo así como: “si no tiene chícharos frescos, utilice congelados”. Esto ilustra otro aspecto de los programas: a menudo llevan a cabo una evaluación y después realizan una de dos tareas, dependiendo del resultado de esa evaluación. A esto se le conoce como selección y, al igual que con la repetición, Java cuenta con herramientas especiales para realizar esta tarea.

Si alguna vez ha utilizado una receta para preparar una comida, es muy probable que haya llegado hasta cierto punto de la receta sólo para descubrir que tiene que consultar otra receta. Por ejemplo, tal vez tenga que ir a otra página para averiguar cómo se cocina el arroz antes de poder combinarlo con el resto de la comida: la preparación del arroz se ha separado como una subtarea. Esta forma de escribir instrucciones tiene una importante analogía en la programación, conocida como métodos en Java y otros lenguajes orientados a objetos. Los métodos se utilizan en todos los lenguajes de programación aunque algunas veces tienen otros nombres, como funciones, procedimientos, subrutinas o subprogramas.

Los métodos son subtareas, y se les llama así debido a que son un método para realizar algo. El uso de métodos ayuda a mantener simples los programas, que de otro modo serían complejos.

Ahora veamos cómo cocinar un curry. Hace algunos años, la receta nos sugeriría comprar especias frescas, molerlas y freírlas. Sin embargo, en la actualidad podemos comprar salsas ya preparadas. Nuestra tarea se ha simplificado. La analogía con la programación consiste en que la tarea se facilita si podemos seleccionar de entre un conjunto de “objetos” prefabricados, como botones, barras de desplazamiento y bases de datos. Java incluye un gran conjunto de objetos que podemos incorporar en nuestro programa en vez de tener que crear todo desde cero.

Para resumir, un programa es una lista de instrucciones que una computadora puede llevar a cabo de manera automática. Un programa consiste en combinaciones de:

- Secuencias.
- Repeticiones.
- Selecciones.
- Métodos.
- Objetos predefinidos.
- Objetos que usted mismo escribe.

Todos los lenguajes de programación modernos comparten estas características.

## PRÁCTICAS DE AUTOEVALUACIÓN

- 1.1** Siga estas instrucciones para calcular el sueldo de un empleado:

```
obtener el número de horas trabajadas
calcular el sueldo
imprimir el recibo de pago
restar las deducciones por enfermedad
```

¿Hay algún error importante?

- 1.2** Modifique la instrucción:

```
dar masaje para incorporar el champú
```

para expresarla de una manera más detallada, incorporando el concepto de repetición.

- 1.3** Las siguientes instrucciones se muestran en un letrero de la montaña rusa:

```
;Sólo pueden subir a este paseo los mayores de 8 años o los menores de
70 años!
```

¿Hay algún problema con el aviso? ¿Cómo podría escribirlo para mejorarlo?

## Fundamentos de programación

- Los programas constan de instrucciones combinadas con los conceptos de secuencia, selección, repetición y subtareas.
- La tarea de programación se simplifica si podemos utilizar componentes predefinidos.

## Errores comunes de programación

Los errores humanos pueden filtrarse en los programas —como colocar las instrucciones en el orden incorrecto.

## Resumen

- Java es un lenguaje orientado a objetos, derivado de C++.
- Los programas de Java son portables: se pueden ejecutar en la mayoría de los tipos de computadoras.
- Java está integrado en los navegadores Web. Los programas tipo applets se pueden ejecutar en los navegadores Web.
- Un programa es una lista de instrucciones que una computadora lleva a cabo de manera automática.
- En la actualidad, la principal tendencia en la práctica de la programación es la metodología de la programación orientada a objetos (POO), y Java se apega a ella en su totalidad.

## Ejercicios

- 1.1** Este ejercicio trata sobre los pasos que realiza un estudiante para levantarse e ir a la escuela. He aquí una sugerencia para los primeros pasos:

```
despertar
vestirse
desayunar
lavarse los dientes
...
```

- (a) Complete los pasos. Observe que no hay una respuesta ideal —los pasos varían de un individuo a otro.
  - (b) El paso “lavarse los dientes” contiene repetición —lo hacemos una y otra vez. Identifique otro paso que contenga repetición.
  - (c) Identifique un paso que contenga una selección.
  - (d) Divida uno de los pasos en varios pasos más pequeños.
- 1.2** Suponga que recibe una enorme pila de papel que contiene 10,000 números, sin ningún orden en especial. Escriba el proceso que realizaría para encontrar el número más grande. Asegúrese de que su proceso sea claro y no tenga ambigüedades. Identifique cualquier selección y repetición en su proceso.
- 1.3** Para el juego del “gato” (tres en raya; ceros y cruces), trate de escribir un conjunto de instrucciones precisas que le permitan ganar a un jugador. Si esto no es posible, trate de asegurar que un jugador no pierda.

## Respuestas a las prácticas de autoevaluación

- 1.1** El principal error es que la parte de las deducciones se lleva a cabo demasiado tarde. Debería ir antes de la impresión.

- 1.2** Podríamos decir:

**siga dando masaje a su cabello hasta que esté completamente enjuagado.**

O:

**mientras su cabello contenga champú, siga dándole masaje.**

- 1.3** El problema está en la palabra 'o'. Alguien que tenga 73 también es mayor de 8 y, por lo tanto, podría subirse a la montaña rusa.

Podríamos sustituir la 'o' por 'y' para que fuera técnicamente correcta, pero aún así la advertencia se podría malentender. También podríamos escribir:

**sólo podrán subirse las personas que tengan entre 8 y 70 años**

pero debe estar preparado para modificar el aviso de nuevo, ¡cuando multitudes de personas de 8 y 70 años pregunten si se pueden subir!

## CAPÍTULO 2



# Los primeros programas

En este capítulo conoceremos:

- Cómo crear, compilar y ejecutar programas de Java.
- El uso de un entorno de desarrollo integrado.
- Los conceptos de clases, objetos y métodos.
- Cómo mostrar un cuadro de diálogo.
- Cómo colocar texto en un campo de texto.

### ● Introducción

Para aprender a programar en Java necesitará una computadora con herramientas de Java, y por fortuna el lenguaje Java está diseñado para ejecutarse en cualquier sistema operativo. En la actualidad los sistemas operativos más utilizados son los sistemas Windows de Microsoft en las PC, aunque también se utilizan los sistemas operativos Linux de GNU en estos equipos, y OS X en las Mac de Apple. Java se puede ejecutar en cualquiera de estos sistemas. Éste es un beneficio importante, pero significa que hay una variación en las instrucciones detalladas para usar Java. Aquí le proporcionaremos la información general solamente. En el apéndice I se incluyen más detalles sobre cómo obtener sistemas de Java sin costo.

Una vez instalado Java, tendrá que atravesar cuatro etapas para generar un programa:

- Crear un nuevo archivo o proyecto.
- Introducir/modificar el programa con un editor.
- Compilar el programa.
- Ejecutar el programa.

## ● Entornos de desarrollo integrados

Hay dos formas principales de crear y ejecutar sus programas. En primer lugar, podríamos elegir un entorno de desarrollo integrado (IDE). Éste es un paquete de software diseñado para ayudar en todo el proceso de crear y ejecutar un programa de Java. Si utiliza un IDE, de todas formas es conveniente que se familiarice con los conceptos de archivos, edición, compilación y ejecución, como veremos más adelante.

Hay varios IDE en la actualidad. Uno de los más populares, además de gratuito, es Eclipse (que también está escrito en Java). En el apéndice I encontrará más información al respecto. Como alternativa puede usar un editor de texto (en vez de un procesador de palabras simple) para crear sus programas. Algunos de los más poderosos (como Textpad) se pueden configurar para enlazar software de Java, para que usted pueda iniciar los procesos de compilación y ejecución haciendo clic en una opción de menú.

## ● Archivos y carpetas

Los programas que se cargan y ejecutan de manera automática al encender la computadora se conocen en forma colectiva como sistema operativo. Una parte importante de un sistema operativo se encarga del almacenamiento de los archivos; a continuación veremos una breve introducción.

La información que se almacena en un disco de computadora se organiza en forma de archivos, al igual que la información guardada en los archiveros de una oficina se almacena en archivos.

Por lo general, un archivo contiene información relacionada. Por ejemplo:

- Una carta para su mamá.
- Una lista de los estudiantes de un curso específico.
- Una lista de amigos con nombres, direcciones y números telefónicos.

Cada archivo tiene su propio nombre, elegido por la persona que lo creó. Como podría esperarse, es común elegir un nombre que describa con claridad lo que hay en el archivo. El nombre del archivo tiene una *extensión* (una parte al final) que describe el tipo de información que contiene. Por ejemplo, un archivo llamado **carta1** que contiene una carta y se edita por lo general con un procesador de texto podría tener la extensión **.doc** (abreviación de documento), de manera que su nombre completo sería **carta1.doc**. Un archivo que contiene un programa de Java tiene la extensión **.java**, por lo que un nombre de archivo común podría ser **Juego.java**.

Un grupo de archivos relacionados se guarda en una carpeta (algunas veces se le conoce también como directorio). Por lo tanto, en una carpeta específica podría guardar todas las cartas enviadas al banco. En otra carpeta podría almacenar todas las cifras de ventas de un año. En definitiva, es conveniente mantener todos los archivos que se utilizan en un programa de Java en la misma carpeta. Puede asignar un nombre a cada carpeta (por lo general un nombre representativo) que le ayude a encontrarla.

En general, las carpetas también se pueden agrupar en otra carpeta. Por ejemplo, podría tener una carpeta llamada **Toms** que contenga a las carpetas **misprogs** y **cartas**.

Tal vez piense que esto puede continuar de manera indefinida, y así es, podemos establecer carpetas de carpetas *ad infinitum*. Comúnmente su sistema de cómputo contendrá cientos de carpetas y miles de archivos. Algunos de éstos serán suyos (puede crearlos y después modificarlos) y otros pertenecerán al sistema operativo (*¡no los modifique!*).

Entonces, un archivo es una colección de información con un nombre. Los archivos relacionados se guardan dentro de una carpeta, la cual también tiene un nombre.

Para poder ver las listas de carpetas y los archivos que contienen, debemos utilizar un programa conocido como *Explorador de Windows* en los sistemas operativos Windows de Microsoft. Al hacer clic en una carpeta, ésta muestra los archivos que contiene. Los sistemas Linux de GNU y Mac de Apple tienen herramientas similares.

## PRÁCTICA DE AUTOEVALUACIÓN

- 2.1** (a) ¿Cuál es la diferencia entre una carpeta y un directorio?  
(b) ¿Qué es una carpeta?  
(c) ¿Es posible crear dos carpetas con el mismo nombre?

## ● Creación de un programa de Java

Ya sea que utilice un poderoso IDE o un editor de texto más sencillo, hay varios pasos necesarios para crear un programa de Java.

### Creación de un nuevo archivo

Si utiliza un editor de texto tiene que crear un nuevo archivo, el cual debe tener la extensión `.java`. En un IDE debe crear un proyecto, el cual consta de varios archivos. El único archivo que debe modificar es el `.java`.

### Edición del archivo

Para ello tiene que escribir y modificar el programa (más adelante le mostraremos un programa de ejemplo). Puede utilizar un editor de texto, pero todos los IDEs traen un componente editor de texto. En el editor pasará mucho tiempo, por lo tanto es importante que explore sus herramientas avanzadas, como las que se utilizan para buscar y reemplazar texto.

### Compilación del programa

Un compilador es un programa que convierte otro programa escrito en un lenguaje como Java en un lenguaje que la computadora pueda entender. Por lo tanto, un compilador es como un traductor automático capaz de traducir un lenguaje (de computadora) a otro. Los programas de Java se convierten a código de bytes. El código de bytes no es exactamente el lenguaje que una computadora comprende (código de máquina). Por el contrario, es un lenguaje de máquina idealizado, lo cual quiere decir que su programa de Java se ejecutará en cualquier tipo de computadora. Al ejecutar su programa, el código de bytes es interpretado por un programa conocido como Máquina Virtual de Java (JVM).

Para iniciar el proceso de compilación debemos hacer clic en un botón o seleccionar una opción de menú.

Al compilar su programa, el compilador de Java verifica que éste se apegue a las reglas de programación de Java y, si algo sale mal, muestra los mensajes de error apropiados. También verifica que los programas de cualquier biblioteca que usted utilice se empleen en forma correcta. Es raro (incluso para los programadores experimentados) que un programa se compile correctamente al primer intento, así que no se desanime si recibe algunos mensajes de error. He aquí un ejemplo de un mensaje de error:

```
Hola.java:9: ';' expected
```

Este mensaje indica el nombre del archivo, el número de línea del error (9 en este caso) y una descripción del mismo. Necesitamos comprender el error (esto no es siempre obvio) y después regresar al editor para corregir el texto.

## Ejecución del programa

Una vez que eliminamos todos los errores de compilación, el programa se puede ejecutar (interpretar). Para iniciar la JVM tiene que hacer clic en un botón o usar una opción de menú. Ahora podemos ver el efecto del programa a medida que hace su trabajo.

Tome en cuenta que en algunos IDE, al hacer clic para iniciar el proceso de compilación también se iniciará la ejecución del programa si no se encuentran errores de compilación.

## PRÁCTICA DE AUTOEVALUACIÓN

- 2.2 (a) Averigüe cómo iniciar y usar su editor o su IDE.
- (b) Escriba en su editor texto que contenga la palabra “el” varias veces. Averigüe cómo puede reemplazar en su editor cada ocurrencia de “el” por “ella” con un solo comando.

## Su primer programa

Use el editor o el IDE para crear un nuevo archivo (llamado `Hola.java`) o proyecto y después introduzca el pequeño programa de Java que mostraremos a continuación.

Un archivo que contenga un programa de Java debe tener la extensión `.java`. El nombre del archivo debe coincidir con el nombre de la clase, que va después de las palabras `public class` en el código de Java. El programador puede elegir este nombre a su gusto, aunque se sugiere que sea un nombre representativo de la clase.

No se preocupe por lo que significa en esta etapa. Como puede ver, el programa contiene ciertos caracteres inusuales y tres tipos distintos de paréntesis. Tal vez tenga que buscarlos en su teclado. El texto que introduzca en su editor o IDE se conoce como el *código* de Java.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Hello extends JFrame {

 public static void main(String[] args) {
 JOptionPane.showMessageDialog(null, "¡Hola mundo!");
 JOptionPane.showMessageDialog(null, "Adiós");
 System.exit(0);
 }
}
```

Sin duda cometerá errores al teclear este programa. Puede usar el editor para corregir el programa.

Cuando el programa se vea correcto, trate de compilarlo y observe si aparecen mensajes de error. Para corregirlos, compare su código con nuestra versión.

Una de las paradojas en la programación es que con frecuencia los mensajes de error de los compiladores son enigmáticos y de poca ayuda. El compilador le indicará (nota: no señalará con precisión) la posición de los errores. Analice lo que introdujo y trate de encontrar lo que está mal. Los errores comunes son:

- Faltan puntos y comas o están en el lugar incorrecto.
- Faltan llaves.
- Utilizó comillas sencillas (') en vez de comillas dobles ("').

Identifique su error, edite el programa y vuelva a compilarlo. ¡Aquí es en donde se pone a prueba su paciencia! Repita el proceso hasta que elimine los errores.

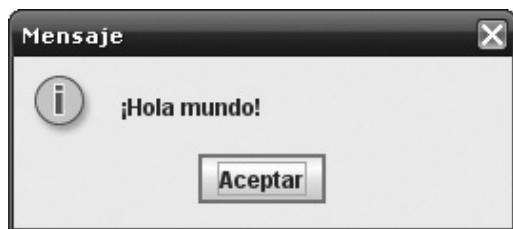
### PRÁCTICA DE AUTOEVALUACIÓN

- 2.3 Cometa un error intencional en su código, borrando un punto y coma. Observe el mensaje de error que produce el compilador. Después vuelva a poner el punto y coma en su lugar.

Después de editar y compilar el programa sin errores, podemos ejecutarlo. El compilador crea un archivo en el disco con la extensión **class**. La primera parte del nombre coincide con el nombre del programa de Java; en este ejemplo el nombre de archivo sería:

**Hola.class**

Éste es el archivo que la JVM ejecutará. Para iniciar el proceso tiene que hacer clic en un botón o seleccionar una opción de menú.



**Figura 2.1** La pantalla de `Hola.java`.

Ahora se ejecutará el programa. Primero mostrará el siguiente mensaje:

`¡Hola mundo!`

Haga clic en **Aceptar** para cerrar este mensaje y que aparezca el siguiente mensaje:

`Adiós`

Haga clic en **Aceptar** de nuevo. El programa terminará. En la figura 2.1 se muestra la pantalla del primer mensaje.

## ● Las bibliotecas

Como vimos, la salida que produce el compilador es un archivo `.class`, el cual podemos ejecutar con la JVM. Sin embargo, el archivo de clase no contiene el programa completo. De hecho, todo programa de Java necesita cierta ayuda de parte de una o más piezas de programas contenidos en bibliotecas. En términos computacionales, una biblioteca es una colección de piezas útiles de programas ya escritos con anterioridad, los cuales se guardan en archivos. Su pequeño programa de ejemplo necesita utilizar una de esas piezas de programas para mostrar información en la pantalla. Para poder lograr esto, la pieza de programa requerida se tiene que enlazar con su programa al momento de ejecutarlo.

Las bibliotecas son colecciones de piezas útiles. Suponga que desea diseñar un nuevo automóvil. Probablemente quiera diseñar la carrocería y los interiores. Pero tal vez quiera usar un motor que alguien más haya diseñado y construido con anterioridad. De manera similar, podría también usar las ruedas que algún otro fabricante haya producido. Así, algunos elementos del automóvil serían nuevos y otros se obtendrían de terceros. Los componentes de terceros son como las piezas de programas en la biblioteca de Java. Por ejemplo, nuestro programa de ejemplo usa un cuadro de diálogo que está contenido en una biblioteca.

Las cosas pueden salir mal a la hora de que el compilador verifique los vínculos con el software de la biblioteca y tal vez emita un mensaje de error críptico. Los errores comunes son:

- Falta la biblioteca o no se encuentra.
- Usted escribió mal el nombre de algo en la biblioteca.

Las bibliotecas se incorporan a su programa cuando éste se ejecuta.

## PRÁCTICA DE AUTOEVALUACIÓN

- 2.4 (a) Busque dos errores en este código:

```
JOptionPane.showMessageDialog(null, "Hola mundo");
```

- (b) ¿Cuál de estos dos errores evitará que el programa se ejecute?

## Desmitificación del programa

Ahora veamos las generalidades sobre el programa de Java. Aun cuando es bastante pequeño, podemos ver que el programa tiene muchos componentes. Esto se debe a que Java es un verdadero lenguaje de uso industrial, e incluso el programa más pequeño necesita algunos ingredientes importantes. Tenga en cuenta que en esta primera etapa no cubriremos todos los detalles. En los siguientes capítulos nos haremos cargo de ello.

En la figura 2.2 le mostraremos de nuevo el código del programa. Esta vez tiene números de línea para ayudar con la explicación (los números de línea no deben formar parte de un programa real). Las líneas 8 y 9 son las piezas más importantes de este programa. Instruyen a la computadora para que muestre texto en un rectángulo desplegable, conocido como cuadro de diálogo, de la clase `JOptionPane`. La línea 8 muestra el texto `;Hola mundo!`, que debe ir encerrado entre comillas dobles. El texto entre comillas de este tipo se denomina cadena de texto, o cadena. La línea termina con un punto y coma, como lo hacen muchas líneas en Java. De manera similar, la línea 9 muestra `"Adiós"`. En Java, la letra `\$` (que representa a Java, desde luego) se antepone a muchos nombres (como `JOptionPane`).

En la figura 2.1 se muestra el efecto de la línea 8. Al hacer clic en Aceptar, el programa continúa con la línea 9. Esta vez se muestra la cadena `"Adiós"`. Observe que los cuadros de diálogo se muestran en secuencia, recorriendo el programa hacia abajo.

```
1 import java.awt.*;
2 import java.awt.event.*;
3 import javax.swing.*;
4
5 public class Hello extends JFrame {
6
7 public static void main(String[] args) {
8 JOptionPane.showMessageDialog(null, ";Hola mundo!");
9 JOptionPane.showMessageDialog(null, "Adiós");
10 System.exit(0);
11 }
12 }
```

Figura 2.2 El programa `Hola`.

## PRÁCTICA DE AUTOEVALUACIÓN

- 2.5 Modifique el programa de manera que muestre un tercer cuadro de diálogo en el que aparezca la cadena “Ahora estoy terminando”.

En la parte superior, las líneas 1, 2 y 3 especifican información sobre los programas de biblioteca que nuestro programa utiliza. La palabra `import` va seguida del nombre de una biblioteca que se utilizará en el programa. Este programa utiliza las bibliotecas AWT (Kit de herramientas de Ventanas Abstractas) y Swing para mostrar un cuadro de diálogo.

La línea 4 está en blanco. Podemos usar líneas en blanco en cualquier parte, para mejorar la legibilidad de un programa.

La línea 5 es un encabezado que anuncia que el código es un programa llamado `Hola`.

El programa en sí va encerrado dentro de las llaves. La { de apertura en la línea 5 coincide con la } de cierre en la línea 12. Dentro de estas líneas hay más llaves. La { en la línea 7 coincide con la } en la línea 11.

La línea 10 hace que el programa se deje de ejecutar.

Más adelante le presentaremos un programa más extenso que muestra texto en la pantalla de una manera distinta. Pero antes de hacerlo, veamos el uso de los “objetos” en Java.

## ● Objetos, métodos: una introducción

Una de las razones de la popularidad de Java es que se trata de un lenguaje *orientado a objetos*. Éste es el tema principal del libro, por lo cual lo veremos con detalle en capítulos posteriores. Pero aquí le presentaremos el concepto de los objetos por medio de una analogía.

En primer lugar imagine una casa con un reproductor de CD en la cocina y uno idéntico en la recámara. En la jerga de Java, los denominamos “objetos”. Ahora considere las herramientas que cada reproductor de CD nos ofrece. En otras palabras, ¿qué botones tiene un reproductor de CD? En la jerga de Java, a cada herramienta se le denomina “método” (por ejemplo, podríamos tener los métodos reproducir y detener, y un método para saltar pistas con base en su número).

El término *método* es algo extraño; proviene de la historia de los lenguajes de programación. Imagine que significa “función” o “herramienta”, como en “Este reproductor de CD tiene un método reproducir y un método detener”.

Ahora considere la tarea de identificar cada botón en cada uno de los reproductores. No basta con decir:

`detener`

ya que hay dos reproductores. ¿A cuál de ellos nos referimos? Lo que debemos hacer es identificar el reproductor también. Si nos basamos en algo más parecido a Java, podríamos usar:

`cdCocina.detener`

Observe el uso del punto. Se utiliza de una forma similar en todos los lenguajes orientados a objetos. Tenemos dos elementos:

- El nombre de un objeto; por lo general corresponde a un sustantivo.
- Un método que provee el objeto; por lo general es un verbo.

Más adelante, cuando hablemos sobre los métodos con más detalle, veremos que la versión exacta de Java para el anterior ejemplo es:

```
cdCocina.detener();
```

Observe el punto y coma y los paréntesis que no contienen nada. Para algunos otros métodos tal vez tengamos que suministrar información adicional con la que el método pueda trabajar, como seleccionar una pista numerada:

```
cdRecamara.seleccionar(4);
```

Al elemento entre los paréntesis se le conoce como “parámetro” (de nuevo tenemos un término de programación tradicional, en vez de uno que tenga significado de inmediato).

En términos generales, la forma en que usamos los métodos es:

```
objeto.método(parámetros);
```

Si el método específico no necesita parámetros, de todas formas debemos usar los paréntesis. En el capítulo 5 veremos los parámetros y los métodos.

## PRÁCTICA DE AUTOEVALUACIÓN

- 2.6** Suponga que nuestros reproductores de CD de la recámara y de la cocina tienen herramientas (métodos) para detener, iniciar y seleccionar una pista numerada. He aquí un ejemplo de cómo usar uno de los métodos:

```
cdCocina.seleccionar(6);
```

Proporcione cinco ejemplos sobre el uso de los métodos.

## Clases: una analogía

El concepto de clase es sumamente importante en la programación orientada a objetos. Recuerde nuestra analogía: tenemos dos objetos reproductores de CD idénticos en nuestra casa. En la jerga de lenguajes orientados a objetos, decimos que tenemos dos “instancias” de la “clase” reproductor de CD. Una clase es como una línea de producción que puede fabricar nuevos reproductores de CD.

Vamos a diferenciar entre una clase y las instancias de una clase. La casa tiene dos instancias de la clase reproductor de CD, las cuales existen de verdad: realmente podemos usarlas. Una clase es un concepto más abstracto. Aunque la línea de producción de reproductores de CD posee el diseño (de una forma u otra) de un reproductor de CD, no existe una instancia real sino hasta que la

máquina fabrica una. En Java utilizamos la palabra “new” para indicarle a una clase que fabrique una nueva instancia. Para resumir:

- Los objetos son instancias de una clase.
- Una clase puede producir todas las instancias que necesitemos.

Vale la pena repetir que estos conceptos son los principales de este libro, por lo cual los cubriremos con mucho más detalle en capítulos posteriores. No esperamos que usted pueda crear programas de Java con objetos y clases en este capítulo.

## Uso de un campo de texto

El primer programa que vimos utilizaba un cuadro de diálogo. El segundo programa que presentaremos es más grande, pero constituye la base para muchos de los programas de este libro. Utiliza un *campo de texto* para mostrar una sola línea de texto, la cadena “¡Hola!” en este caso. En la figura 2.3 se muestra la pantalla y en la figura 2.4 aparece el código con números de línea.

En esta etapa necesitamos recordarle de nuevo que empezaremos a ver los detalles sobre Java en el siguiente capítulo. Por ahora le vamos a mostrar algunos programas con una explicación general sobre lo que hacen. No esperamos (todavía) que usted pueda analizar una línea de código y decir con precisión lo que hace.

Como antes, incluimos los números de línea para que ayuden en nuestras explicaciones, pero no debe introducirlos junto con el programa. El nombre que sigue a las palabras `public class` es **Saludo**, por lo que debe guardar el programa en un archivo llamado:

```
Saludo.java
```

Compile y ejecute el programa. Para detenerlo, haga clic en la cruz en la esquina superior derecha de la ventana o haga clic en el icono de Java en la esquina superior izquierda y después seleccione **Cerrar** en el menú.

Ahora veremos algunos de los usos de instancias, métodos y parámetros.

Recuerde nuestras analogías. Anteriormente mencionamos el uso de la notación “punto” para los objetos y sus métodos asociados. Localice la línea 11:

```
marco.setSize(300, 200);
```



Figura 2.3 La pantalla de `Saludo.java`.

```
1 import java.awt.*;
2 import java.awt.event.*;
3 import javax.swing.*;
4
5 public class Saludo extends JFrame {
6
7 private JTextField textField;
8
9 public static void main (String[] args) {
10 Saludo marco = new Saludo();
11 marco.setSize(300, 200);
12 marco.crearGUI();
13 marco.setVisible(true);
14 }
15
16 private void createGUI() {
17 setDefaultCloseOperation(EXIT_ON_CLOSE);
18 Container window = getContentPane();
19 ventana.setLayout(new FlowLayout());
20 campoTexto = new JTextField("¡Hola!");
21 ventana.add(campoTexto);
22 }
23 }
```

Figura 2.4 El programa **Saludo**.

Aquí se utiliza la misma notación: objeto, punto, método, parámetros.

Imagine el objeto marco como el borde exterior de la pantalla de la figura 2.3. El método **setSize** recibe dos parámetros: la anchura y altura requeridas del marco en unidades conocidas como píxeles. Aquí utilizamos una metodología orientada a objetos para establecer el tamaño del marco.

## PRÁCTICA DE AUTOEVALUACIÓN

2.7 Modifique el programa **Saludo** de manera que el marco tenga la mitad de la anchura.

He aquí otro uso de los objetos. Localice las siguientes líneas en el programa:

```
campoTexto = new JTextField("¡Hola!");
ventana.add(campoTexto);
```

En primer lugar, se crea un campo de texto mediante la palabra **new**. En esta etapa podemos elegir el texto que aparecerá en el campo de texto, aunque el usuario podrá sobrescribirlo a la hora de ejecutar el programa. Después se agrega el campo de texto al objeto ventana. Cuando se ejecuta el programa aparece el campo de texto y se centra de manera automática.

## PRÁCTICAS DE AUTOEVALUACIÓN

- 2.8 Modifique el programa para que muestre el siguiente texto:

Un bloque de texto en un campo de texto

- 2.9 Ejecute el programa **Saludo** y trate de cambiar el tamaño del marco arrastrándolo con el ratón. ¿Qué ocurre con la posición del campo de texto?

Por último, una observación general sobre nuestro segundo programa. La mayor parte de las instrucciones tienen que ver con la especificación de las bibliotecas necesarias y cómo establecer la apariencia visual de la pantalla, es decir, la “interfaz gráfica de usuario” o GUI.

Imagine la GUI de su procesador de texto favorito. En la parte superior de su ventana podrá ver una gran cantidad de menús y botones. Al ejecutar el procesador de texto los menús y botones aparecerán en forma instantánea, por lo que tal vez se sorprenda al saber que, tras bambalinas, el procesador de texto empieza con una ventana toda en blanco y laboriosamente agrega cada menú y botón a la ventana, uno por uno. Debido a la velocidad de la computadora el proceso parece instantáneo.

Al escribir programas más grandes, de todas formas se tiene que realizar la configuración inicial de la pantalla, pero esa parte del código se vuelve menos importante que el código encargado de que el programa realice una tarea al momento de que el usuario hace clic en un botón o en un elemento de menú.

## Fundamentos de programación

- Una de las principales características de Java es el uso extendido de las clases.
- La notación “punto” para usar los objetos es:  
`objeto.método(parámetros)`
- Las instrucciones se llevan a cabo en secuencia, desde la parte superior del programa hasta la parte inferior.

## Errores comunes de programación

- Cuando edite un programa, guárdealo cada 10 minutos aproximadamente para evitar perder su trabajo en caso de que falle la computadora.
- Al escribir un programa, asegúrese de copiar los caracteres con exactitud; use las mayúsculas según se muestra.
- Asegúrese de que el nombre del archivo coincida con el nombre de la clase en el mismo. Por ejemplo:

```
public class Hola extends JFrame {
```

Aquí, el nombre que va después de **class** es **Hola**. Hay que guardar el archivo como **Hola.java**. La letra mayúscula de la clase es importante.

- Es muy probable que cometa errores al escribir un programa. El compilador le indicará cuáles son los errores. Procure no frustrarse demasiado por los errores.

## Secretos de codificación

Los programas de Java contienen varias llaves de apertura y cierre. Debe haber el mismo número de llaves de cierre que de llaves de apertura.

## Nuevos elementos del lenguaje

Un cuadro de diálogo puede mostrar una cadena de texto junto con un botón **Aceptar**, como en:

```
JOptionPane.showMessageDialog(null, "¡Hola mundo!");
```

## Resumen

- Los programas de Java se pueden crear y ejecutar en la mayoría de los tipos de computadoras.
- Se utiliza un editor o un IDE para crear y modificar el código fuente de los programas de Java.
- Hay que compilar un programa de Java antes de poder ejecutarlo.
- Es necesario corregir los errores de compilación para poder ejecutar un programa.
- El compilador produce un archivo con el mismo nombre que el archivo original de Java, pero con la extensión **class**. Este archivo contiene instrucciones en código de bytes.
- La JVM (Máquina Virtual de Java) se utiliza para ejecutar programas.
- Gran parte del poder de Java proviene de sus bibliotecas, las cuales se enlazan al momento de ejecutar el programa.
- Java es orientado a objetos. Utiliza los conceptos de clases, instancias y métodos.
- La notación “punto” se utiliza en todos los programas de Java. He aquí un ejemplo:

```
marco.setSize(300, 200);
```

- Los métodos (como **setSize**) realizan tareas sobre el objeto especificado (como **marco**).
- Las cosas pueden salir mal en cualquier etapa; parte del trabajo del programador es identificar y corregir los errores. No lo olvide: es extraño que todo funcione sin problemas la primera vez. Tenga cuidado y relájese.

## Ejercicios

- 2.1** Asegúrese de saber cómo compilar y ejecutar programas de Java en su computadora. Compile y ejecute los dos programas de este capítulo.
- 2.2** En el programa **Hola**, agregue un cuadro de diálogo para mostrar su nombre.
- 2.3** En el programa **Saludo**, haga que el campo de texto muestre su nombre.

## Respuestas a las prácticas de autoevaluación

- 2.1** (a) No hay diferencia. Los términos significan lo mismo.  
(b) Una carpeta contiene varios archivos y/u otras carpetas.  
(c) Sí. Mientras las dos carpetas con nombres idénticos no estén dentro de la misma carpeta (por ejemplo, cada una de las carpetas **Trabajo** y **Casa** podrían contener una carpeta llamada **cartas**).
- 2.2** (a) Esto depende de los editores disponibles en su computadora. Si utiliza un IDE, ya incluye un editor.  
(b) Esto depende de su editor. Muchos editores tienen una herramienta **Buscar...Reemplazar**, la cual explora todo el texto.
- 2.3** El mensaje de error variará, dependiendo del punto y coma que haya omitido.
- 2.4** (a) Hay una “p” incorrecta. Debe ser “P” como en **JOptionPane**. La palabra **mindo** está mal escrita.  
(b) El error de la “p” evitara que el programa se compile, por lo tanto no se podrá ejecutar. El error “mindo” no evitara que el programa se ejecute, pero el resultado no será el esperado.
- 2.5** Inserte la siguiente línea justo después de la línea 9, que muestra el texto **“Adiós”**:

```
JOptionPane.showMessageDialog(null, "Ahora estoy terminando");
```

Compile y ejecute el programa modificado.

- 2.6**

```
cdCocina.reproducir();
cdCocina.detener();
cdRecamara.reproducir();
cdRecamara.detener();
cdRecamara.seleccionar(3);
```

- 2.7** Reemplace el **300** por **150** en la línea 11, después vuelva a compilar el programa y ejecútelo.

- 2.8** Reemplace el texto **“;Hola!”** en la línea 20 por:

```
"Un bloque de texto en un campo de texto"
```

Compile y ejecute el programa.

- 2.9** Permanece centrado cerca de la parte superior del marco.

# CAPÍTULO 3



## Uso de métodos gráficos

En este capítulo conoceremos:

- La naturaleza de los eventos.
- Cómo dibujar figuras con métodos gráficos.
- El uso de los parámetros.
- Cómo comentar programas.
- Cómo usar colores.
- El concepto de secuencia.

### ● Introducción

El término *gráficos de computadora* evoca una variedad de posibilidades. Podríamos estar hablando de una película de Hollywood generada por computadora, de una fotografía estática o de una imagen más simple, construida a partir de líneas. En este capítulo nos limitaremos a las imágenes estáticas elaboradas a partir de formas simples, y nos enfocaremos en el uso de métodos de biblioteca para crearlas. También introduciremos el uso de un botón para permitir la interacción del usuario.

### ● Eventos

Muchos programas se crean de cierta forma para permitir la interacción del usuario mediante una GUI (Interfaz Gráfica de Usuario). Dichos programas proveen botones, campos de texto, barras de desplazamiento, etc. En términos de Java, cuando el usuario manipula el ratón y el teclado crea “eventos” a los que el programa responde. Algunos eventos comunes son:

- El clic de del ratón.
- Oprimir una tecla.
- El uso de un control deslizable para desplazarnos por ciertos valores.

El sistema de Java clasifica los eventos en varias categorías. Por ejemplo, desplazarse por una página se considera un evento de “cambio”, mientras que hacer clic en un botón se considera como un evento de “acción”.

Al escribir un programa de Java, debe asegurarse de que el programa detecte los eventos; de lo contrario no ocurrirá nada. La transmisión de un evento (como un clic del ratón) a un programa no se lleva a cabo en forma automática; debemos configurar el programa para que “detecte” los tipos de eventos. Por fortuna, el código para ello es estándar y lo podemos reutilizar en todos los programas, en vez de tener que crearlo de nuevo para cada programa.

Al proceso de responder a un evento se le conoce como “manejar” el evento.

El siguiente programa provee un botón. La figura 3.1 muestra la pantalla.



**Figura 3.1** Pantalla del programa `EjemploDibujo` después de hacer clic en el botón.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class EjemploDibujo extends JFrame
 implements ActionListener {

 private JButton button;
 private JPanel panel;
```

```
public static void main(String[] args) {
 EjemploDibujo marco = new EjemploDibujo();
 marco.setSize(400, 300);
 marco.createGUI();
 marco.setVisible(true);
}

private void createGUI() {
 setDefaultCloseOperation(EXIT_ON_CLOSE);
 Container ventana = getContentPane();
 ventana.setLayout(new FlowLayout());

 panel = new JPanel();
 panel.setPreferredSize(new Dimension(300, 200));
 panel.setBackground(Color.white);
 ventana.add(panel);

 boton = new JButton("Haz clic");
 ventana.add(boton);
 boton.addActionListener(this);
}

public void actionPerformed(ActionEvent event) {
 Graphics papel = panel.getGraphics();
 papel.drawLine(0, 0, 100, 100);
}
```

El usuario hace clic en el botón y se dibuja una línea diagonal. Para los fines de este capítulo, nos interesa principalmente la siguiente instrucción:

```
papel.drawLine(0, 0, 100, 100);
```

Como es de esperarse, esta instrucción es la que dibuja la línea. El resto del código configura la GUI, y hablaremos de él sólo en términos generales. Para configurar la GUI se requiere:

- Agregar un botón a la ventana, de una manera similar a la que agregamos un campo de texto en el capítulo 2.
- Agregar un “panel” que utilizaremos para dibujar.
- Preparar el programa para que detecte los clic del ratón (clasificados como eventos de acción).

El siguiente punto es muy importante: en capítulos posteriores veremos la creación de interfaces de usuario. Por ahora consideraremos el código anterior de la GUI como estándar.

## ● El evento de clic de botón

El principal evento en este programa se genera cuando el usuario hace clic en el botón **“Haz clic”**. Un clic de botón hace que el programa ejecute esta sección del programa:

```
public void actionPerformed(ActionEvent event) {
 Graphics papel = panel.getGraphics();
 papel.drawLine(0, 0, 100, 100);
}
```

Esta sección del programa es un *método*, conocido como `actionPerformed`. Cuando el usuario hace clic en el botón, el sistema de Java invoca al método y se ejecutan en secuencia las instrucciones entre la llave de apertura (`{`) y la de cierre (`}`). Aquí es en donde colocamos nuestras instrucciones de dibujo. Ahora veremos los detalles sobre el dibujo de figuras.

## ● El sistema de coordenadas de gráficos

Los gráficos de Java se basan en píxeles. Un píxel es un pequeño punto en la pantalla que se puede cambiar a un color específico. Cada píxel se identifica mediante un par de números (sus coordenadas), empezando desde cero:

- La posición horizontal, que a menudo se denomina *x* en las matemáticas (y también en la documentación de Java); este valor se incrementa de izquierda a derecha.
- La posición vertical, que a menudo se denomina *y*; este valor se incrementa hacia abajo. Tome en cuenta que esto es distinto a la convención en matemáticas.

Utilizamos este sistema cuando pedimos a Java que dibuje formas simples. La figura 3.2 muestra la metodología. La parte superior izquierda del área de dibujo es (0, 0) y el trazo de las formas comienza a partir de este punto.

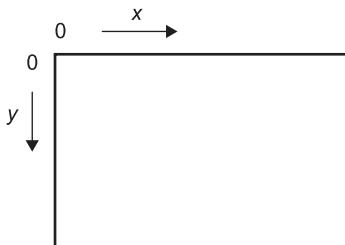


Figura 3.2 El sistema de coordenadas de píxeles.

## ● Explicación del programa

La única sección que nos concierne es la parte pequeña que hace el dibujo:

```
1 public void actionPerformed(ActionEvent event) {
2 Graphics papel = panel.getGraphics();
3 papel.drawLine(0, 0, 100, 100);
4 }
```

La línea 1 introduce la sección del programa que se ejecuta al hacer clic en el botón. Es un método. Cualquier instrucción que coloquemos entre el carácter `{` de la línea 1 y el carácter `}` de la línea 4 se ejecutará en secuencia, descendiendo por la página.

La línea 2 provee un área de gráficos para dibujar figuras; en este caso la llamamos `papel`. Si recuerda, en el capítulo 2 mencionamos que Java es orientado a objetos. Los objetos nos proporcionan herramientas. En el capítulo 2 vimos un ejemplo de un reproductor de CD, el cual provee un rango de herramientas como la siguiente:

```
cdCocina.seleccionar(4);
```

De hecho, nuestra área de dibujo no es sólo una hoja de papel en blanco; es más como un kit de dibujo que incluye el papel y un conjunto de herramientas, como una regla y un transportador.

En la línea 3 el papel utiliza su método `drawLine` para dibujar una línea sobre sí mismo. Los cuatro números entre paréntesis especifican la posición de la línea.

El método `drawLine` es uno de los diversos métodos que proporciona el sistema Java en una biblioteca. La línea 3 es una llamada (también conocida como invocación) al método, en donde le pide que realice la tarea de mostrar una línea en pantalla.

Al utilizar el método `drawLine` tenemos que suministrarle valores para los puntos inicial y final de la línea; además, necesitamos hacerlo en el orden correcto:

1. El valor horizontal (*x*) del inicio de la línea.
2. El valor vertical (*y*) del inicio de la línea.
3. El valor horizontal del final de la línea.
4. El valor vertical del final de la línea.

En Java estos elementos se denominan parámetros; son entradas para el método `drawLine`. Los parámetros deben ir encerrados entre paréntesis y separados por comas (tal vez se encuentre con el término *argumento*, que es otro nombre para un parámetro). Este método en especial requiere cuatro parámetros, los cuales deben ser números enteros. Si tratamos de usar el número incorrecto de parámetros, o que sean de tipo incorrecto, recibiremos un mensaje de error del compilador. Necesitamos asegurarnos de:

- Proveer el número correcto de parámetros.
- Proveer el tipo correcto de parámetros.
- Ordenarlos de la forma correcta.

Algunos métodos no requieren parámetros. En este caso también debemos utilizar los paréntesis, como en:

```
marco.crearGUI();
```

En nuestro ejemplo hay dos tipos de métodos en acción:

- Los que escribe el programador, como `actionPerformed`. El sistema Java invoca este método cuando el usuario hace clic en el botón.
- Los que se incluyen en las bibliotecas, como `drawLine`. Nuestro programa los invoca.

Una última observación: observe el punto y coma ‘;’ al final de los parámetros de `drawLine`. En Java debe aparecer un punto y coma al final de cada “instrucción”. Pero, ¿qué es una instrucción? ¡La respuesta no es tan sencilla! Como puede ver en el programa anterior, no hay un punto y coma al final de cada línea. En vez de confundirlo con reglas formales complicadas aquí, le aconsejamos

que base sus primeros programas en nuestros ejemplos. Sin embargo, el uso de un método seguido de sus parámetros es de hecho una instrucción, por lo que se requiere un punto y coma.

## Métodos para dibujar

Además de líneas, la biblioteca de Java nos proporciona herramientas para dibujar:

- Rectángulos.
- Óvalos (y, por ende, círculos).

A continuación le mostraremos una lista de los parámetros para cada método, junto con un programa de ejemplo en el que se utilizan.

### **drawLine**

- El valor horizontal del inicio de la línea.
- El valor vertical del inicio de la línea.
- El valor horizontal del final de la línea.
- El valor vertical del final de la línea.

### **drawRect**

- El valor horizontal de la esquina superior izquierda.
- El valor vertical de la esquina superior izquierda.
- La anchura del rectángulo.
- La altura del rectángulo.

### **drawOval**

Imagine el óvalo ajustado dentro de un rectángulo. Los parámetros que requerimos son:

- El valor horizontal de la esquina superior izquierda del rectángulo.
- El valor vertical de la esquina superior izquierda del rectángulo.
- La anchura del rectángulo.
- La altura del rectángulo.

También se pueden dibujar las siguientes figuras, pero se requiere un conocimiento adicional sobre Java. Vamos a omitir los detalles sobre sus parámetros, ya que no las usaremos en nuestros programas.

- Arcos (sectores de un círculo).
- Rectángulos elevados (tridimensionales).
- Rectángulos con esquinas redondeadas.
- Polígonos.

Además, podemos dibujar figuras sólidas con `fillRect` y `fillOval`. Sus parámetros son idénticos a los de los métodos `drawRect` y `drawOval` equivalentes.

## Dibujos a color

Es posible establecer el color para dibujar. Hay 13 colores estándar:

|        |       |           |          |
|--------|-------|-----------|----------|
| black  | blue  | cyan      | darkGray |
| gray   | green | lightGray | magenta  |
| orange | pink  | red       | white    |
| yellow |       |           |          |

(el `cyan` es verde oscuro/azul y el `magenta` es rojo oscuro/azul).

Tenga cuidado con la ortografía; tome en cuenta el uso de mayúsculas a la mitad de algunos nombres. He aquí cómo podemos usar los colores:

```
papel.setColor(Color.red);
papel.drawLine(0, 0, 100, 50);
papel.setColor(Color.green);
papel.drawOval(100, 100, 50, 50);
```

El código anterior dibuja una línea roja, después un óvalo verde sin relleno. Si no establece el color, Java elige el negro.

## Creación de un nuevo programa

En el programa `EjemploDibujo` anterior nos concentraremos en su método `actionPerformed`, el cual contenía una invocación al método `drawLine`. Nuestro enfoque era aprender sobre la invocación y el paso de parámetros a los métodos de dibujo. Pero, ¿qué hay sobre las demás líneas de código? Están involucradas en tareas tales como:

- Crear el marco exterior (ventana) para el programa.
- Establecer el tamaño del marco.
- Agregar el área de dibujo y el botón a la interfaz de usuario.

Estas tareas se llevan a cabo mediante la invocación a métodos. En capítulos posteriores le explicaremos los detalles.

De hecho, la configuración de la interfaz de usuario es idéntica para cada programa de este capítulo. Todos los programas utilizan un área de dibujo y un solo botón para iniciar el dibujo. Sin embargo, no podemos usar el código idéntico en cada programa ya que el nombre de archivo a elegir debe coincidir con el nombre después de las palabras `public class` dentro del programa. Dé un vistazo al programa `EjemploDibujo`. Está almacenado en un archivo llamado `EjemploDibujo.java` y contiene la siguiente línea:

```
public class EjemploDibujo extends JFrame
```

El nombre de la clase debe empezar con letra mayúscula y sólo puede contener letras y dígitos. No puede contener signos de puntuación, como guiones, comas ni puntos; tampoco puede contener espacios. Al elegir el nombre de la clase se determina el nombre del archivo que debemos usar para guardar el programa.

Hay una línea adicional:

```
EjemploDibujo marco = new EjemploDibujo();
```

Esta línea se encuentra dentro del método `main` del programa. Al ejecutar un programa de Java, lo primero que ocurre es una invocación automática al método `main`. La primera tarea de `main` es crear una nueva instancia (un objeto) de la clase apropiada (en este caso, `EjemploDibujo`). En el capítulo 6 examinaremos el uso de `new` para crear nuevas instancias. Por ahora, tome en cuenta que el programa `EjemploDibujo` contiene tres ocurrencias del nombre `EjemploDibujo`:

- Una después de las palabras `public class`.
- Dos en el método `main`.

Al crear un nuevo programa tiene que cambiar estas ocurrencias.

He aquí un ejemplo. Vamos a crear un nuevo programa llamado `CirculoDibujo`. Los pasos son:

1. Abrir el archivo `EjemploDibujo.java` existente en cualquier editor, seleccionar todo el texto y copiarlo al portapapeles.
2. Abrir el software que utiliza para crear y ejecutar programas de Java (un IDE tal como Eclipse o un editor de texto).
3. Si utiliza un IDE, tendrá que crear un nuevo proyecto en este punto. El nombre del proyecto podría ser `ProyectoCirculoDibujo` (el nombre no necesita estar relacionado con `CirculoDibujo`, pero facilita la identificación de los proyectos).
4. Cree un nuevo archivo de Java en blanco llamado `CirculoDibujo.java`. Si utiliza un IDE, tal vez tenga que eliminar código que el IDE crea de manera automática.
5. Pegue el código que copió y cambie las tres ocurrencias de `EjemploDibujo` por `CirculoDibujo`.

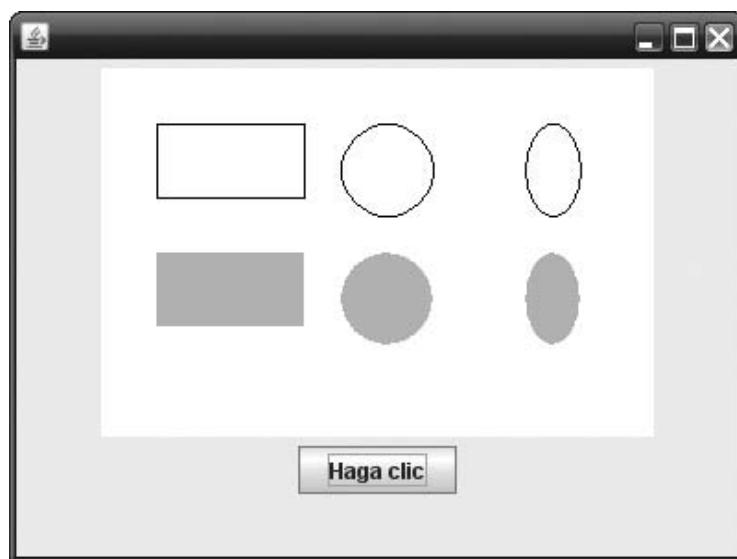
Ahora se puede concentrar en el tema principal de este capítulo y colocar las invocaciones apropiadas a los métodos de dibujo dentro del método `actionPerformed`.

## PRÁCTICA DE AUTOEVALUACIÓN

- 3.1 Cree un nuevo programa llamado `CirculoDibujo`. Al hacer clic en el botón, debe dibujar un círculo de 100 píxeles de diámetro.

### ● El concepto de secuencia

Cuando tenemos varias instrucciones en un programa, éstas se ejecutan de arriba abajo en secuencia (a menos que especifiquemos lo contrario mediante los conceptos de selección y repetición que veremos más adelante). El siguiente programa dibuja diversas figuras. La figura 3.3 muestra la salida resultante. En el listado omitimos el código que crea la interfaz de usuario, ya que esta parte es exactamente igual que en el programa anterior.



**Figura 3.3** Pantalla del programa `AlgunasFiguras`.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class AlgunasFiguras extends JFrame
 implements ActionListener {

 // aquí se omitió el código de la GUI...

 public void actionPerformed(ActionEvent event) {
 Graphics papel = panel.getGraphics();
 papel.drawRect(30, 30, 80, 40);
 papel.drawOval(130, 30, 50, 50);
 papel.drawOval(230, 30, 30, 50);
 papel.setColor(Color.lightGray);
 papel.fillRect(30, 100, 80, 40);
 papel.fillOval(130, 100, 50, 50);
 papel.fillOval(230, 100, 30, 50);
 }
}
```

Las instrucciones se obedecen (ejecutan, llevan a cabo, ...) de arriba hacia abajo por la página, aunque esto es imposible de observar debido a la velocidad de la computadora. En capítulos posteriores veremos cómo repetir una secuencia de instrucciones una y otra vez.

## PRÁCTICA DE AUTOEVALUACIÓN

- 3.2 Escriba y ejecute un programa que dibuje una figura de ‘T’ grande en la pantalla.

### Cómo agregar significado mediante comentarios

¿Qué hacen las siguientes líneas de código?

```
papel.drawLine(20, 80, 70, 10);
papel.drawLine(70, 10, 120, 80);
papel.drawLine(20, 80, 120, 80);
```

El significado no es obvio de inmediato, y probablemente usted haya tratado de averiguarlo utilizando lápiz y papel. La respuesta es que dibuja un triángulo con una base horizontal, pero esto no se puede deducir con sólo ver las tres instrucciones. En Java podemos agregar comentarios (un tipo de anotación) a las instrucciones si les anteponemos los caracteres `//`. Por ejemplo, podríamos poner:

```
// dibuja un triángulo
papel.drawLine(20, 80, 70, 10);
papel.drawLine(70, 10, 120, 80);
papel.drawLine(20, 80, 120, 80);
```

Un comentario puede contener cualquier cosa —no hay reglas definidas—. Es responsabilidad de usted utilizarlos para expresar cierto significado.

También podemos colocar comentarios al final de una línea, como en el siguiente ejemplo:

```
// dibuja un triángulo
papel.drawLine(20, 80, 70, 10);
papel.drawLine(70, 10, 120, 80);
papel.drawLine(20, 80, 120, 80); // dibuja la base
```

No es conveniente abusar de los comentarios. No se recomienda comentar cada una de las líneas de un programa, ya que a menudo se corre el riesgo de duplicar información. El siguiente es un ejemplo de un mal comentario:

```
papel.drawRect(0, 0, 100, 100); // dibuja un rectángulo
```

Aquí la instrucción indica con claridad lo que hace, sin necesitar un comentario. Use los comentarios para declarar el tema general de una sección del programa, en vez de recalcar los detalles de cada instrucción. En el capítulo 19 aprenderá acerca de los estilos de comentarios adicionales que ayudan en la documentación de un programa.

## Fundamentos de programación

- Java cuenta con un vasto conjunto de métodos de biblioteca que podemos invocar.
- Los parámetros que pasamos a los métodos tienen el efecto de controlar las figuras que se dibujan.

## Errores comunes de programación

- Tenga cuidado con la puntuación. Las comas, los puntos y comas, los paréntesis y las llaves se deben escribir exactamente como se muestra en los ejemplos.
- El uso de mayúsculas debe ser exacto. Por ejemplo, `drawline` está mal, mientras que `drawLine` es correcto.

## Secretos de codificación

El orden y tipo de los parámetros debe ser correcto para cada método.

## Nuevos elementos del lenguaje

- ( ) para encerrar una lista de parámetros separados por comas.
- Si no se requieren parámetros, es necesario colocar los caracteres () después del nombre del método.
- // para indicar comentarios.

## Resumen

- Los programas pueden detectar eventos.
- Responder a un evento se conoce como “manejar” el evento.
- Las instrucciones se obedecen en secuencia, de arriba abajo (a menos que se solicite lo contrario).
- La biblioteca de Java tiene un conjunto de métodos “draw” que podemos invocar para mostrar gráficos.
- El posicionamiento de los gráficos se basa en coordenadas de píxeles.
- Se pueden pasar valores de parámetros a los métodos.

## Ejercicios

En los siguientes ejercicios le recomendamos que realice bosquejos y cálculos antes de escribir el programa. Seleccione un nombre adecuado para cada programa, que empiece con una letra mayúscula y no contenga espacios ni signos de puntuación. Base su programa en el código de `EjemploDibujo`, e inserte nuevas instrucciones dentro del método `actionPerformed`.

- 3.1** (a) Dibuje un cuadrado con tamaño de **100** píxeles, que comience a **10** píxeles de la esquina superior izquierda del área de dibujo. Use **drawRect**.  
(b) Realice la misma tarea, pero llame a **drawLine** cuatro veces.
- 3.2** Dibuje un triángulo con un lado vertical.
- 3.3** Dibuje un tablero para el juego del “gato” (o tres en línea).
- 3.4** Diseñe una casa sencilla y dibújela.
- 3.5** Dibuje una paleta de colores que conste de 13 pequeños cuadrados, cada uno de los cuales debe contener un color distinto.
- 3.6** He aquí las cifras de precipitaciones pluviales anuales para el país de Xanadú:

|      |        |
|------|--------|
| 2004 | 150 cm |
| 2005 | 175 cm |
| 2006 | 120 cm |

- (a) Represente los datos mediante una serie de líneas horizontales.  
(b) En vez de líneas utilice rectángulos rellenos de distintos colores.
- 3.7** Diseñe una diana con círculos concéntricos. Después agregue distintos colores.

## Respuestas a las prácticas de autoevaluación

- 3.1** Abra el programa **EjemploDibujo** existente con un editor. Guárdelo con el nombre **CirculoDibujo.java**. Cambie las tres ocurrencias de **EjemploDibujo** por **CirculoDibujo**; asegúrese de que el uso de mayúsculas sea el correcto. Por último, llame al método **drawOval** con los parámetros apropiados en el método **actionPerformed**. He aquí el programa completo:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class CirculoDibujo extends JFrame
 implements ActionListener {

 private JButton button;
 private JPanel panel;

 public static void main(String[] args) {
 CirculoDibujo marco = new CirculoDibujo();
 marco.setSize(400, 300);
 marco.createGUI();
 marco.setVisible(true);
 }
}
```





Respuestas a las prácticas de autoevaluación (continúa)

```
private void createGUI() {
 setDefaultCloseOperation(EXIT_ON_CLOSE);
 Container ventana = getContentPane();
 ventana.setLayout(new FlowLayout());

 panel = new JPanel();
 panel.setPreferredSize(new Dimension(300, 200));
 panel.setBackground(Color.white);
 ventana.add(panel);

 boton = new JButton("Haga clic");
 ventana.add(boton);
 boton.addActionListener(this);
}

public void actionPerformed(ActionEvent event) {
 Graphics papel = panel.getGraphics();
 papel.drawOval(0, 0, 100, 100);
}
}

3.2 papel.drawLine(20, 20, 120, 20);
papel.drawLine(70, 20, 70, 120);
```

# CAPÍTULO 4



## Variables y cálculos

En este capítulo conoceremos:

- Los tipos de variables numéricas.
- Cómo declarar variables y constantes.
- La instrucción de asignación.
- Los operadores aritméticos.
- El uso de cuadros de mensaje y de entrada para las operaciones de entrada y salida.
- Los aspectos esenciales sobre las cadenas de texto.

### ● Introducción

En la mayoría de los programas se utilizan números de un tipo u otro; por ejemplo, para dibujar imágenes mediante coordenadas en la pantalla, controlar trayectorias de vuelos espaciales, así como calcular sueldos y deducciones fiscales.

En este capítulo veremos los dos tipos básicos de números:

- Los números enteros, conocidos como tales en las matemáticas y como tipo `int` en Java.
- Los números con “punto decimal”, conocidos como “reales” en las matemáticas y como tipo `double` en Java. El término general para los números con punto decimal en computación es *números de punto flotante*.

En el capítulo anterior utilizamos valores para producir gráficos en pantalla, pero para programas más sofisticados necesitamos introducir el concepto de variable: una especie de caja de almacenamiento que se utiliza para recordar valores, de forma que éstos se puedan utilizar o modificar más adelante en el programa.

Algunos casos indiscutibles en los que se utilizan números `int` son:

- El número de estudiantes en una clase.
- El número de píxeles en la pantalla.
- El número de copias de este libro vendidas hasta ahora.

Y también hay casos indiscutibles en los que se utilizan números `double`:

- Mi altura en metros.
- La masa de un átomo en gramos.
- El promedio de los números enteros 3 y 4.

Sin embargo, algunas veces el tipo no es obvio; por ejemplo, una variable para almacenar la calificación de un examen: ¿`double` o `int`? No podemos determinar la respuesta con lo que sabemos hasta el momento; debemos buscar más detalles. Por ejemplo, podemos preguntar a la persona que se encarga de calificar si redondea al número entero más cercano, o si utiliza lugares decimales. Por ende, la elección entre `int` y `double` se determina de acuerdo con el problema.

## ● La naturaleza de `int`

Cuando utilizamos un número `int` en Java, puede ser un número entero en el rango de  $-2147483648$  a  $+2147483647$ , o aproximadamente de  $-2000000000$  a  $+2000000000$ .

Todos los cálculos con números `int` son precisos en cuanto a que toda la información en el número se preserva con exactitud.

## ● La naturaleza de `double`

Cuando utilizamos un número `double` en Java, su valor puede estar entre  $-1.79 \times 10^{308}$  y  $+1.79 \times 10^{308}$ .

En términos no tan matemáticos, el valor más grande es 179 seguido de 306 ceros; ¡sin duda un valor extremadamente grande! Los números se guardan con una precisión aproximada de 15 dígitos.

El principal detalle sobre las cantidades `double` es que en muchos casos se guardan en forma aproximada. Realice la siguiente operación en una calculadora:

7 / 3

Si utilizamos siete dígitos (por ejemplo), la respuesta es 2.333333, pero sabemos que una respuesta más exacta sería:

2.333333333333333

Y aun así, ¡esta no es la respuesta exacta!

En resumen, como las cantidades `double` se almacenan en un número limitado de dígitos, se pueden acumular pequeños errores en el extremo menos significativo. Para muchos cálculos (por ejemplo, calificaciones de exámenes) esto no es importante, pero para cálculos relacionados con el diseño de un transbordador espacial esto sí podría ser significativo. Sin embargo, los números `double`

tienen tantos dígitos de precisión y un rango tan extenso que pueden utilizarse sin problemas en los cálculos relacionados con las cantidades que manejamos cotidianamente.

Para escribir valores **double** muy grandes (o muy pequeños) se requieren grandes secuencias de ceros. Para simplificar estas cifras podemos usar la notación “científica” o “exponencial” con **e** o **E**, como en el siguiente ejemplo:

```
double valorGrande = 12.3E+23;
```

lo cual representa a un 12.3 multiplicado por  $10^{+23}$ . Esta característica se utiliza principalmente en programas matemáticos o científicos.

## ● Declaración de variables

Una vez elegido el tipo de nuestras variables, necesitamos darles un nombre. Podemos imaginarlas como cajas de almacenamiento con un nombre en su exterior y un número (valor) en su interior. El valor puede cambiar a medida que el programa realiza su secuencia de operaciones, pero el nombre es fijo. El programador puede elegir los nombres, y recomendamos elegir nombres que sean significativos y no enigmáticos. Pero al igual que en la mayoría de los lenguajes de programación, hay ciertas reglas que debemos seguir. En Java, los nombres:

- Deben empezar con una letra (de la “A” a la “Z” o de la “a” a la “z”) o (lo que no es muy común) con un guión bajo “\_”.
- Pueden contener cualquier cantidad de letras o dígitos (un dígito es del 0 al 9).
- Pueden contener el guión bajo “\_”.
- Pueden tener hasta 255 caracteres de longitud.

Tome en cuenta que Java es susceptible al uso de mayúsculas y minúsculas. Por ejemplo, si usted declara una variable llamada **anchura**, no se puede referir a ella como **Anchura** o **ANCHURA**, ya que el uso de las mayúsculas y minúsculas es distinto en cada caso.

Éstas son las reglas de Java y debemos obedecerlas. Pero Java también tiene un estilo, una forma de usar las reglas que se implementa cuando una variable consta de varias palabras. Las reglas no permiten espacios en los nombres, por lo que en vez de utilizar nombres cortos o el guión bajo, el estilo aceptado para las variables es poner en mayúscula la primera letra de cada palabra dentro de un nombre.

Hay otro lineamiento de estilo para decidir si usar mayúscula en la primera letra de un nombre o no. Todas las variables deben empezar con una letra minúscula, mientras que los nombres de las clases (como veremos más adelante) empiezan por lo general con una letra mayúscula. Por lo tanto, en vez de:

```
Alturadecaja
a
adc
altura_de_caja
```

usaremos:

```
alturaDeCaja
```

He aquí algunos nombres permitidos:

```
cantidad
x
pago2003
```

y he aquí algunos nombres no permitidos (ilegales):

```
2001pago
%área
mi edad
```

También existen algunos nombres reservados que Java utiliza que el programador no puede reutilizar. Se denominan *palabras clave* o *palabras reservadas* en Java. Ya ha visto algunas de estas palabras clave:

```
private
new
int
```

El apéndice F incluye una lista completa.

## PRÁCTICA DE AUTOEVALUACIÓN

- 4.1 ¿Cuáles de los siguientes nombres de variables locales están permitidos en Java y cuáles tienen el estilo correcto?

```
volumen
ÁREA
Longitud
3lados
lado1
lonitud
Misalario
su salario
tamañoPantalla
screenSize
```

A continuación estudiaremos con detalle el siguiente programa de ejemplo, llamado **CálculoÁrea**. Este programa calcula el área de un rectángulo. Vamos a suponer que sus lados son cantidades **int**. El resultado se muestra en un cuadro de mensaje. La parte más importante se muestra con una pantalla.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class CálculoÁrea extends JFrame
implements ActionListener {

 private JButton botón;

 public static void main(String[] args) {
 CálculoÁrea marco = new CálculoArea();
 marco.setSize(400, 300);
 marco.crearGUI();
 marco.setVisible(true);
 }
}
```

```
private void createGUI() {
 setDefaultCloseOperation(EXIT_ON_CLOSE);
 Container ventana = getContentPane();
 ventana.setLayout(new FlowLayout());

 botón = new JButton("Haga clic");
 ventana.add(botón);
 botón.addActionListener(this);
}

public void actionPerformed(ActionEvent event) {
 int área;
 int longitud;
 int anchura;
 longitud = 20;
 anchura = 10;
 área = longitud * anchura;
 JOptionPane.showMessageDialog(null, "El área es: " + área);
}
```



La figura 4.1 muestra la ventana y el cuadro de mensaje que aparecen al ejecutar el programa.



**Figura 4.1** Pantalla de la salida de `CalculoArea` en donde se muestra la ventana y el cuadro de mensaje.

Para nuestros fines en este capítulo, el programa consta de dos partes. La primera es responsable de crear una ventana que contiene un solo botón. Esto será estándar para todos los programas de este capítulo. La parte que examinaremos con detalle tiene el siguiente encabezado:

```
public void actionPerformed(ActionEvent event) {
```

Esta sección se ejecuta cada vez que el usuario hace clic en el botón. De hecho, la sección se conoce como “método” y debemos colocar nuestras instrucciones entre sus llaves de apertura y de cierre.

En el programa utilizamos tres variables `int`, que en un momento dado guardarán los datos de nuestro rectángulo. Recuerde que podemos elegir cualquier nombre para nuestras variables, sin embargo optamos por utilizar nombres legibles en vez de nombres graciosos o de una sola letra (¡los nombres graciosos sólo son graciosos la primera vez que los vemos!).

Una vez elegidos los nombres, debemos declararlos en el sistema de Java. Aunque esto parece tedioso al principio, el objeto de introducirlos es permitir que el compilador detecte errores ortográficos en el código del programa en el que se utilicen estos nombres. He aquí las declaraciones:

```
int área;
int longitud;
int anchura;
```

Para declarar variables anteponemos al nombre que elegimos el tipo que necesitamos (en las tres variables anteriores utilizamos el tipo `int`, de manera que cada variable puede contener un número entero).

La siguiente línea puede ser una alternativa a las tres líneas anteriores de código:

```
int longitud, anchura, área;
```

aquí usamos comas para separar cada nombre. Usted puede utilizar el estilo de su preferencia, pero nosotros preferimos usar el primero ya que nos permite insertar comentarios en cada nombre en caso de ser necesario. Si usted opta por el segundo estilo, úselo para agrupar nombres relacionados. Por ejemplo, use:

```
int alturaImagen, anchuraImagen;
int miEdad;
```

en vez de:

```
int alturaImagen, anchuraImagen, miEdad;
```

En la mayoría de los programas utilizaremos varios tipos, y en Java podemos entremezclar las declaraciones, como en el siguiente ejemplo:

```
double alturaPersona;
int calificaciónExamen;
double salario;
```

Además, podemos optar por inicializar el valor de la variable al tiempo que la declaramos, como en:

```
double alturaPersona = 1.68;
int a = 3, b = 4;
int calificaciónExamen = 65;
int mejorCalificación = calificaciónExamen + 10;
```

Este es un buen estilo, pero sólo debe usarlo cuando conozca el valor inicial. Si no suministra un valor inicial para las variables declaradas dentro de un método, Java las considera como no inicializadas y un error de compilación le informará sobre ello si trata de usarlas en el programa.

## ● La instrucción de asignación

Una vez declaradas nuestras variables, podemos colocar nuevos valores en ellas mediante la “instrucción de asignación”, como en el siguiente ejemplo:

```
longitud = 20;
```

Podemos visualizar el proceso como se muestra en la figura 4.2. Decimos que “el valor 20 se ha asignado a la variable `longitud`”, o que “`longitud` se vuelve 20”.

Nota:

- El movimiento de los datos es de la derecha del signo = hacia la izquierda.
- Cualquier valor que haya tenido `longitud` antes será “sustituido” por el 20. Las variables sólo tienen un valor: el actual. Y para darle una idea de la velocidad de estas operaciones, una asignación tarda menos de una millonésima parte de un segundo.

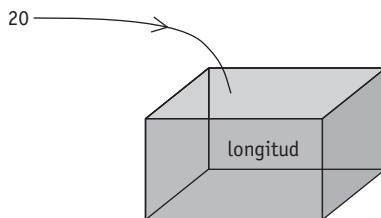


Figura 4.2 Cómo asignar un valor a una variable.

## PRÁCTICA DE AUTOEVALUACIÓN

**4.2** Explique el problema con este fragmento de código:

```
int a, b;
a = b;
b = 1;
```

## ● Cálculos y operadores

Veamos de nuevo nuestro programa del rectángulo, en el que se incluye la siguiente instrucción:

```
área = longitud * anchura;
```

La forma general de la instrucción de asignación es:

```
variable = expresión;
```

Una expresión puede tomar varias formas, como un solo número o un cálculo. En nuestro ejemplo específico, la secuencia de eventos es:

1. El signo `*` hace que se multipliquen los valores almacenados en `longitud` y `anchura`, y se obtiene como resultado el valor 200.
2. El signo `=` hace que el 200 se asigne a (se almacene en) `área`.

El signo `*` es uno de varios “operadores” (se les llama así debido a que operan sobre los valores) y, al igual que en matemáticas, hay reglas para su uso.

Es importante comprender el movimiento de los datos, ya que nos permite comprender el significado de código tal como:

```
int n = 10;
n = n + 1;
```

Lo que ocurre aquí es que el lado derecho del signo `=` se calcula utilizando el valor actual de `n`, con lo cual se obtiene un 11. Después este valor se almacena en `n`, sustituyendo al valor anterior de 10.

Hace algunos años se estudiaba una gran cantidad de programas, y se descubrió que las instrucciones de la forma:

```
algo = algo + 1;
```

¡eran de las más comunes! De hecho, Java cuenta con una versión abreviada de esta instrucción de *incremento*. Los operadores `++` y `--` realizan el incremento y decremento (restar 1). Su uso más frecuente es en los ciclos (capítulo 8). He aquí una forma de utilizar el operador `++`:

```
n = 3;
n++; // ahora n vale 4
```

Lo importante sobre el signo `=` es que no significa “es igual a” en el sentido algebraico. Debemos imaginar que significa “se convierte en” o “recibe”.

## ● Los operadores aritméticos

En esta sección le presentaremos un conjunto básico de operadores: los aritméticos, similares a los botones de su calculadora.

| Operador       | Significado    |
|----------------|----------------|
| <code>*</code> | multiplicación |
| <code>/</code> | división       |
| <code>%</code> | módulo         |
| <hr/>          |                |
| <code>+</code> | suma           |
| <code>-</code> | resta          |

Observe que dividimos a los operadores en grupos para indicar su “precedencia”: el orden en el que se llevan a cabo. Por lo tanto, `*`, `/` y `%` se llevan a cabo antes que `+` y `-`. También podemos

usar paréntesis para agrupar los cálculos y forzarlos a que se calculen primero. Si un cálculo incluye operadores de la misma precedencia, se lleva a cabo de izquierda a derecha. He aquí algunos ejemplos:

```
int i;
int n = 3;
double d;
i = n + 3; // se convierte en 6
i = n * 4; // se convierte en 12
i = 7 + 2 * 4; // se convierte en 15
n = n * (n + 2) * 4; // se convierte en 60
d = 3.5 / 2; // se convierte en 1.75
n = 7 / 4; // se convierte en 1
```

Recuerde que las instrucciones forman una secuencia, la cual se ejecuta de arriba hacia abajo en la página. Siempre que se utilicen paréntesis, los elementos dentro de ellos se calcularán primero. La multiplicación y la división se realizan antes de la suma y la resta. Por lo tanto:

3 + 2 \* 4

se lleva a cabo como si se hubiera escrito así:

3 + (2 \* 4)

Más adelante le explicaremos los detalles sobre los operadores / y %.

Por cuestión de estilo, observe que escribimos un espacio antes y después de un operador. Esto no es esencial puesto que el programa se ejecutará de todas formas si se omiten los espacios. Los utilizamos para que el programa sea más legible para el programador. Además, las líneas se pueden volver demasiado extensas para la pantalla. En este caso insertamos una nueva línea en una parte conveniente (aunque no a la mitad de un nombre) y aplicamos sangría a la segunda parte de la línea.

## PRÁCTICA DE AUTOEVALUACIÓN

4.3 En el siguiente fragmento de código, ¿cuáles son los valores finales de las variables?

```
int a, b, c, d;
d = -8;
a = 1 * 2 + 3;
b = 1 + 2 * 3;
c = (1 + 2) * 3;
c = a + b;
d = -d;
```

Ahora conocemos las reglas. Pero aún hay obstáculos para el principiante. Veamos a continuación algunas fórmulas matemáticas y su conversión a Java. Vamos a suponer que todas las variables están declaradas como tipos **double** y que fueron inicializadas.

| Versión de matemáticas        | Versión de Java                  |
|-------------------------------|----------------------------------|
| 1 $y = mx + c$                | $y = m * x + c;$                 |
| 2 $x = (a - b)(a + b)$        | $x = (a - b) * (a + b);$         |
| 3 $y = 3[(a - b)(a + b)] - x$ | $y = 3 * ((a - b)*(a + b)) - x;$ |
| 4 $y = 1 - \frac{2a}{3b}$     | $y = 1 - (2 * a) / (3 * b);$     |

En el ejemplo 1 insertamos el símbolo de multiplicación. En Java, `mx` se consideraría un nombre de variable.

En el ejemplo 2 necesitamos un signo de multiplicación explícito entre los paréntesis.

En el ejemplo 3 sustituimos los corchetes matemáticos con paréntesis.

En el ejemplo 4 podríamos haber cometido el error de usar esta versión incorrecta:

```
y = 1 - 2 * a / 3 * b
```

Recuerde la regla de izquierda a derecha para los operadores de igual precedencia. El problema tiene que ver con los operadores `*` y `/`. El orden de evaluación es como si hubiéramos utilizado:

```
y = 1 - (2 * a / 3) * b
```

es decir, la `b` ahora está multiplicando en vez de dividir. La forma más simple de manejar los cálculos potencialmente confusos es utilizar paréntesis adicionales; no hay ningún castigo en términos de tamaño o reducción en la velocidad del programa.

El uso de los operadores `+`, `-` y `*` es razonablemente intuitivo, pero la división es un poco más engañosa, ya que necesitamos diferenciar entre los tipos `int` y `double`. Los puntos importantes son:

- Cuando el operador `/` trabaja con dos números `double` o con una mezcla de `double` e `int`, se produce un resultado `double`. Tras bambalinas, cualquier valor `int` se considera como `double` para los fines del cálculo. Así es como funciona la división en una calculadora de bolsillo.
- Cuando el operador `/` trabaja con dos enteros, se produce un resultado entero. El resultado se trunca, lo cual significa que se borran los dígitos después del “punto decimal”. Esta **no** es la forma en que funciona una calculadora.

He aquí algunos ejemplos:

```
// división con valores double
double d;
d = 7.61 / 2.1; // se convierte en 3.62
d = 10.6 / 2; // se convierte en 5.3
```

En el primer caso, la división se lleva a cabo de la forma esperada. En el segundo caso, el número `2` se trata como `2.0` (es decir, un `double`) y la división se lleva a cabo.

Sin embargo, la división con enteros es distinta:

```
/división con enteros
int i;
i = 10 / 5; // se convierte en 2
i = 13 / 5; // se convierte en 2
i = 33 / 44; // se convierte en 0
```

En el primer caso la división con enteros da el resultado esperado. Se produce la respuesta exacta de 2. En el segundo caso el resultado es 2, debido a que se trunca el verdadero resultado. En el tercer caso se trunca la respuesta “correcta” de 0.75, con lo cual obtenemos un 0.

## PRÁCTICAS DE AUTOEVALUACIÓN

- 4.4 Mi salario es de \$20,000 y estoy de acuerdo en darle a usted la mitad del mismo, para lo cual utilizaré el siguiente cálculo:

```
int mitad = 20000 * (1 / 2);
```

¿Cuánto recibirá usted?

- 4.5 Indique los valores con los que terminan **a**, **b**, **c** y **d** después de realizar los siguientes cálculos:

```
int a, b, c, d;
a = 7 / 3;
b = a * 4;
c = (a + 1) / 2;
d = c / 3;
```

### El operador %

Por último, veremos el operador % (módulo). A menudo se utiliza junto con la división de enteros, ya que provee la parte del residuo. Su nombre proviene del término *módulo* que se utiliza en una rama de las matemáticas conocida como aritmética modular.

Anteriormente dijimos que los valores **double** se almacenan en forma aproximada, y que los enteros se almacenan en forma exacta. Entonces ¿cómo puede ser que  $3/4$  genere un resultado entero de 0? ¿Acaso perder el 0.75 significa que el cálculo no es preciso? La respuesta es que los enteros **sí** operan con exactitud, pero la respuesta exacta está compuesta de dos partes: el cociente (es decir, la respuesta principal) y el residuo. Por lo tanto, si dividimos 4 entre 3 obtenemos una respuesta de 1, con un residuo de 1. Esto es más exacto que 1.3333333 etc.

Por ende, el operador % nos da el residuo como si se hubiera llevado a cabo una división. He aquí algunos ejemplos:

```
int i;
double d;
i = 12 % 4; // se convierte en 0
i = 13 % 4; // se convierte en 1
i = 15 % 4; // se convierte en 3
d = 14.9 % 3.9; // se convierte en 3.2 (se divide 3 veces)
```

Cabe mencionar que el operador % también funciona con números **double**, aunque el uso más frecuente de % es con tipos **int**. Veamos ahora un problema con un residuo: convertir un número

entero de centavos en dos cantidades —el número de dólares y el número de centavos restantes. La solución es:

```
int centavos = 234;
int dólares, centavosRestantes;
dólares = centavos / 100; // se convierte en 2
centavosRestantes = centavos % 100; // se convierte en 34
```

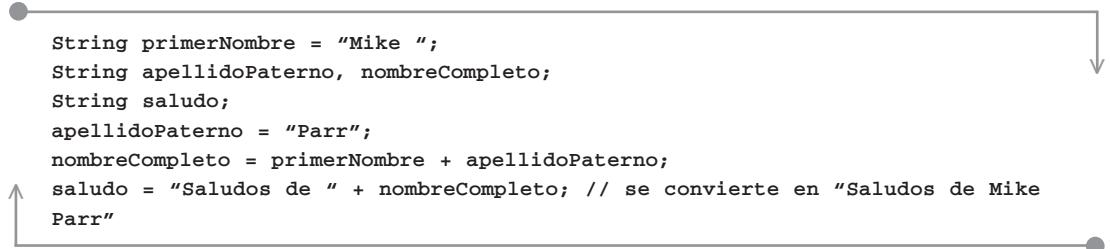
## PRÁCTICA DE AUTOEVALUACIÓN

- 4.6** Complete el siguiente fragmento de código. Agregue instrucciones de asignación para dividir `totalSegundos` en dos variables: `minutos` y `segundos`.

```
int totalSegundos = 307;
```

### Unión de cadenas con el operador +

Hasta ahora hemos visto el uso de variables numéricas, pero también es muy importante el procesamiento de datos de texto. Java cuenta con el tipo de datos `String`; las variables `String` pueden guardar cualquier número de caracteres. He aquí un ejemplo del uso de cadenas:



```
String primerNombre = "Mike ";
String apellidoPaterno, nombreCompleto;
String saludo;
apellidoPaterno = "Parr";
nombreCompleto = primerNombre + apellidoPaterno;
saludo = "Saludos de " + nombreCompleto; // se convierte en "Saludos de Mike
Parr"
```

En el ejemplo anterior declaramos algunas variables `String` y les asignamos valores iniciales mediante el uso de comillas dobles. Observe la `S` mayúscula en `String`.

Después utilizamos la asignación, en la cual el valor de la cadena a la derecha del signo `=` se almacena en la variable utilizada a la izquierda del signo `=`, de forma similar a la asignación numérica.

Las siguientes líneas ilustran el uso del operador `+`, que (al igual que al sumar números) opera sobre las cadenas y las une extremo con extremo. A esto se le conoce como “concatenación”. Después de la instrucción:

```
nombreCompleto = primerNombre + apellidoPaterno;
```

el valor de `nombreCompleto` es `Mike Parr`.

Además hay un amplio rango de métodos de cadenas que proporcionan operaciones tales como búsqueda y modificación de cadenas. Hablaremos sobre estos métodos en el capítulo 15.

Anteriormente mencionamos que el operador `/` considera a los elementos que divide como números `double` si uno de ellos es `double`. El operador `+` trabaja en forma similar con las cadenas. He aquí un ejemplo:

```
int i = 7;
String nombre = "a. avenida";
String s = i + nombre;
```

En este ejemplo el operador `+` detecta que `nombre` es una variable `String` y convierte el valor de `i` en una cadena antes de unir ambas variables. Éste es un método abreviado conveniente para evitar la conversión explícita que veremos a continuación. Pero puede ser engañoso. Considere el siguiente código:

```
int i = 2, j = 3;
String s, nota = "La respuesta es: ";
s = nota + i + j;
```

¿Cuál es el valor de `s`? Las dos posibilidades son:

- La respuesta es: 23, en donde ambos operadores `+` trabajan sobre cadenas.
- La respuesta es: 5, en donde el segundo `+` suma números.

De hecho, el primer caso es el que ocurre. Java trabaja de izquierda a derecha. El primer `+` produce la cadena “La respuesta es: 2”. Después el segundo `+` agrega el 3 a la derecha. No obstante, si colocamos:

```
s = nota + (i + j);
```

primero se calcula la operación `2 + 3` y se obtiene un 5. Por último se lleva a cabo la unión de las cadenas.

Sin embargo, la mayoría de los casos son simples, como en nuestro programa del área:

```
JOptionPane.showMessageDialog(null,"El área es: " + área);
```

El cuadro de mensaje (que presentamos en el capítulo 2) puede mostrar una cadena. En el ejemplo anterior, el cuadro de mensaje muestra una cadena entre comillas unida con la cadena que representa a un entero. La alternativa es que podemos hacer la conversión explícita utilizando el método `toString` como se muestra a continuación.

## PRÁCTICA DE AUTOEVALUACIÓN

- 4.7 Los cuadros de mensaje pueden mostrar una cadena de texto. ¿Qué muestran en pantalla los siguientes cuadros de mensaje?

```
JOptionPane.showMessageDialog(null,
 "5" + "5" + 5 + 5);
JOptionPane.showMessageDialog(null,
 "5" + "5" + (5 + 5));
```

## ● Conversiones entre cadenas y números

Uno de los usos más importantes del tipo de dato `String` es en las operaciones de entrada y salida, en donde procesamos los datos que introduce el usuario y mostramos los resultados en la pantalla. Muchas de las clases de GUI de Java trabajan con cadenas de caracteres en vez de números, por lo cual necesitamos saber cómo realizar conversiones entre números y cadenas.

Para convertir un cálculo o una variable numérica (una expresión en general) en una cadena, podemos utilizar los métodos `toString` de las clases `Integer` y `Double`. He aquí algunos ejemplos:

```

String s1, s2;
int num = 44;
double d = 1.234;
s1 = Integer.toString(num); // s1 es "44"
s2 = Double.toString(d); // s2 es "1.234"

```

Por lo general, el nombre de un método (como `toString`) va precedido de un objeto con el que debe trabajar, pero aquí suministramos el nombre de una clase (`Integer` o `Double`). Los métodos que funcionan de esta forma se denominan estáticos (`static`); cada vez que los utilicemos debemos identificar la clase a la que pertenecen. Las clases `Double` e `Integer` contienen herramientas adicionales para los tipos `double` e `int`. Tome en cuenta que (como siempre) los nombres de las clases empiezan con mayúscula. En el capítulo 9 veremos los métodos estáticos.

Para convertir cadenas en números utilizamos los métodos “parse” de las clases `Double` e `Integer`. El término *parse* se utiliza en el sentido de explorar un bloque de texto para examinarlo. A diferencia de los métodos que utilizamos en el capítulo 2 (como `drawLine`), estos métodos devuelven un valor cuando se les invoca y podemos guardar el valor devuelto en una variable, o podemos usarlo de alguna otra forma. He aquí algunos ejemplos que muestran cómo se puede convertir una cadena de texto en un valor `int` o `double`:



```

int i;
double d;
String s1 = "1234";
String s2 = "1.23";
i = Integer.parseInt(s1);
d = Double.parseDouble(s2);

```

Los métodos `parseInt` y `parseDouble` requieren un parámetro, el cual debe ser de tipo `String`. Podríamos proveer una cadena de texto entre comillas, una variable o una expresión de cadena de texto.

Como veremos más adelante, el usuario puede haber escrito la cadena de texto y, por ende, podría contener caracteres que no se permitan en los números. Los métodos `parseInt` y `parseDouble` detectarán esos errores y el programa terminará. En el capítulo 16 veremos cómo puede un programa detectar esos errores (“excepciones”) y pedir al usuario que vuelva a escribir un número. Por ahora vamos a suponer que el usuario siempre introducirá datos correctos.

## PRÁCTICA DE AUTOEVALUACIÓN

- 4.8** En el siguiente código, ¿cuáles son los valores finales de `m`, `n` y `s`?

```

int m, n;
String s;
String v = "3";
m = Integer.parseInt(v + v + "4");
n = Integer.parseInt(v + v) + 4;
s = Integer.toString(Integer.parseInt(v)
+ Integer.parseInt(v)) + "4";

```

## Cuadros de mensaje y de entrada

Aquí vamos a analizar los cuadros de diálogo con más detalle. En nuestro programa para calcular el área de un rectángulo, utilizamos un cuadro de mensaje para mostrar el valor del área. Este valor debe estar en forma de una cadena de texto. En vez de sólo mostrar el número, lo unimos en un mensaje:

```
JOptionPane.showMessageDialog(null, "El área es: " + área);
```

Recuerde nuestra descripción anterior sobre el operador `+` al aplicarlo a una cadena y un número: el número se convierte de manera automática en una cadena, por lo que no necesitamos usar `toString`. Sin embargo, podríamos haberlo convertido en forma explícita, como en el siguiente ejemplo:

```
int n = 33;
JOptionPane.showMessageDialog(null,
 "n es: " + Integer.toString(n));
```

Algunas veces tal vez tengamos que mostrar el número sin texto que lo acompañe. Tenga en cuenta que el siguiente código no se compilará debido a que el método `showMessageDialog` espera un valor `String` como parámetro:

```
JOptionPane.showMessageDialog(null, área); // NO - ;no se compilará!
```

En vez de eso, debe usar:

```
JOptionPane.showMessageDialog(null, Integer.toString(área));
```

Por cierto, el primer parámetro para un cuadro de mensaje siempre será `null` en nuestros ejemplos. Esta palabra clave de Java hace que el cuadro de mensaje se posicione en el centro de la pantalla. También se puede posicionar sobre una ventana específica, pero no mostraremos esa posibilidad aquí.

La clase `JOptionPane` también provee un cuadro de entrada, el cual permite al usuario escribir una cadena. He aquí algunos ejemplos:

```
String primerNombre, apellidoPaterno;
primerNombre = JOptionPane.showInputDialog(
 "Escriba su primer nombre");
apellidoPaterno = JOptionPane.showInputDialog(
 "Escriba su apellido paterno");
```

El único parámetro del método `showInputDialog` es una indicación, que se utiliza para informar al usuario sobre los datos requeridos. Al hacer clic en “Aceptar”, la cadena se devuelve al programa para poder asignarla a una variable.

Ahora regresemos a nuestro programa del área. En realidad, es poco probable que conozcamos las medidas del rectángulo al momento de escribir el programa. Vamos a enmendar nuestro programa para que solicite las medidas al momento de ejecutarse. He aquí el programa modificado (en la figura 4.3 se muestra la pantalla del primer cuadro de entrada):

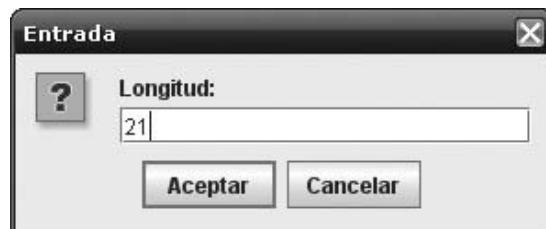


Figura 4.3 Pantalla de un cuadro de entrada del programa CuadrosDiálogoÁrea.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class CuadrosDiálogoÁrea extends JFrame
 implements ActionListener {
 private JButton botón;

 public static void main(String[] args) {
 CuadrosDiálogoÁrea marco = new CuadrosDiálogoÁrea();
 marco.setSize(400, 300);
 marco.createGUI();
 marco.setVisible(true);
 }

 private void createGUI() {
 setDefaultCloseOperation(EXIT_ON_CLOSE);
 Container ventana = getContentPane();
 ventana.setLayout(new FlowLayout());
 botón = new JButton("Haga clic");
 ventana.add(botón);
 botón.addActionListener(this);
 }

 public void actionPerformed(ActionEvent event) {
 int área;
 int longitud;
 int anchura;
 String longitudString;
 String anchuraString;
 longitudString = JOptionPane.showInputDialog("Longitud:");
 longitud = Integer.parseInt(longitudString);
 anchuraString = JOptionPane.showInputDialog("Anchura:");
 anchura = Integer.parseInt(anchuraString);
 área = longitud * anchura;
 JOptionPane.showMessageDialog(null, "El área es: " + área);
 }
}
```

Recuerde que los cuadros de entrada proveen cadenas de texto al programa, las cuales se deben convertir en enteros mediante el método `parseInt`.

## ● Aplicación de formato al texto en cuadros de diálogo mediante \n

Al mostrar texto, a menudo es conveniente mostrarlo como varias líneas cortas en vez de una sola línea extensa. Para ello utilizamos un par especial de caracteres: el carácter “diagonal inversa” seguido de una `n`. Esta combinación, a la que comúnmente se le conoce como nueva línea, se encarga de separar la línea. La combinación `\n` se debe usar como parte del mensaje, entre comillas. He aquí un ejemplo. El programa se llama `CuadrosDiálogoDólares` y muestra los resultados de dividir una cantidad de centavos en dólares enteros y centavos restantes. Sólo mostramos el método `actionPerformed`. Observe la salida en la figura 4.4 y compárela con el uso de `\n` en el cuadro de mensaje.

```
public void actionPerformed(ActionEvent event) {
 int totalCentavos;
 int dólares;
 int centavosRestantes;
 String totalCentavosString;

 totalCentavosString = JOptionPane.showInputDialog(
 "Escriba su monto, en centavos");
 totalCentavos = Integer.parseInt(totalCentavosString);
 dólares = totalCentavos / 100;
 centavosRestantes = totalCentavos % 100;
 JOptionPane.showMessageDialog(null,
 totalCentavosString + " centavos se dividen en:\n" +
 dólares + " dólares\n" +
 centavosRestantes + " centavos.");
}
```

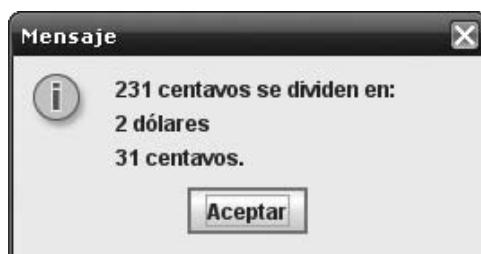


Figura 4.4 Salida del programa `CuadrosDiálogoDólares`.

## ● Conversiones entre números

Algunas veces necesitamos convertir valores numéricos de un tipo a otro. Los casos más comunes son la conversión de un `int` a un `double` y de un `double` a un `int`.

Veamos un ejemplo: tenemos nueve manzanas y queremos compartirlas de manera equitativa entre cuatro personas. Sin duda los valores `9` y `4` son enteros, pero la respuesta es un valor `double`. Para resolver este problema debemos conocer algunos fundamentos sobre la conversión numérica.

Antes de proporcionarle la respuesta, veamos algunos ejemplos de conversiones:

```
int i = 33;
double d = 3.9;
double d1;
d1 = i; // se convierte en 33.0
// o, de manera explícita:
d1 = (double)i; // se convierte en 33.0
i = (int)d; // se convierte en 3
```

Los puntos importantes son:

- Para asignar un `int` a un `double` no se requiere de programación adicional. Es un proceso seguro ya que no se puede perder información; no hay números decimales por los cuales preocuparse.
- Al asignar un `double` a un `int` se pueden perder los dígitos después del punto decimal, ya que no caben en el entero. Debido a esta pérdida potencial de información, Java requiere que especifiquemos esta conversión de manera explícita mediante la conversión de tipos o *casting* (utilizaremos también esta característica cuando veamos las herramientas más avanzadas de la programación orientada a objetos).
- Para convertir un `double` a la forma de un `int`, debemos anteponer la palabra `(int)`. El valor se trunca, eliminando los dígitos después del punto decimal.
- Cabe mencionar que podríamos usar una conversión explícita de tipos al convertir un `int` en un `double`, pero esto no es necesario.

Volviendo a nuestro ejemplo de las manzanas, podemos obtener una respuesta `double` si utilizamos las siguientes líneas de código:

```
int manzanas = 9; // o podemos obtener el valor de un cuadro de texto
int personas = 4; // o podemos obtener el valor de un cuadro de texto
 JOptionPane.showMessageDialog(null, "Una persona recibe: " +
 Double.toString((double)manzanas / (double)personas));
```

Tome en cuenta que `(double)(manzanas / personas)` produciría la respuesta incorrecta, ya que se realizaría una división entre enteros antes de la conversión de tipos.

## PRÁCTICA DE AUTOEVALUACIÓN

**4.9** ¿Cuáles son los valores de `a`, `b`, `c`, `i`, `j` y `k` después de ejecutar el siguiente código?

```
int i, j, k;
double a, b, c;
int n = 3;
double y = 2.7;
i = (int)y;
j = (int)(y + 0.6);
k = (int)((double)n + 0.2);
a = n;
b = (int)n;
c = (int)y;
```

### Constantes: uso de `final`

Hasta ahora, hemos usado variables; es decir, elementos cuyos valores cambian a medida que se ejecuta el programa. Pero algunas veces tenemos valores que nunca cambian. He aquí un ejemplo:

```
double millas, km = 4.7;
millas = km * 0.6214;
```

El significado de esta línea no está claro. ¿Qué es lo que representa `0.6214`? De hecho, es el número de millas en un kilómetro, y el valor nunca cambia.

He aquí cómo podemos reformular el programa mediante la palabra clave `final` de Java:

```
final double millasPorKm = 0.6214;
double millas, km = 4.7;
millas = km * millasPorKm;
```

La palabra `final` indica que ahora la variable especificada tiene su valor final fijo, el cual no puede cambiar cuando el programa se ejecuta. En otras palabras, `millasPorKm` es una constante. Las constantes pueden ser `double`, `int`, `String` o `boolean`.

Cabe mencionar que algunos programadores usan mayúsculas para las constantes (como `MILLASPORKM`), para poder distinguirlas claramente de las variables.

Si usted escribe código (ya sea en forma intencional o accidental) para modificar la constante, se produce un error de compilación, como en el siguiente ejemplo:

```
millasPorKm = 2.1; // error de compilación
```

Además de que usted puede declarar sus propias constantes, las bibliotecas de Java también proveen constantes matemáticas. Por ejemplo, la clase `Math` provee un valor para `pi`, que se puede usar de esta forma:

```
double radio = 1.4, circunferencia;
circunferencia = 2 * Math.PI * radio;
```

Los beneficios de utilizar constantes son:

- Mejora la legibilidad del programa al utilizar nombres en vez de números.
- El uso de constantes minimiza los errores de escritura, ya que al usar repetidas veces un número largo en un programa se pueden cometer errores. Por ejemplo, podríamos cometer un error al escribir `0.6124` en vez de `0.6214`. Dichos errores son difíciles de detectar. En cambio, si escribimos `millasPorKm` como `millasBorKm` se producirá un error de compilación y podremos corregir la ortografía.

## PRÁCTICA DE AUTOEVALUACIÓN

- 4.10** Hay 2.54 cm en una pulgada. Declare una constante llamada `cmPorPulg` con el valor correcto. Muestre cómo se podría usar en un cálculo para convertir pulgadas en centímetros.

### ● El papel que desempeñan las expresiones

Aunque hemos recalcado que las expresiones (cálculos) pueden formar parte del lado derecho de las instrucciones de asignación, también pueden estar en otros lugares. De hecho, podemos colocar una expresión `int` en cualquier parte en donde podamos colocar un valor `int` individual. Considere el método `drawLine` que vimos en ejemplos anteriores, el cual requiere cuatro enteros para especificar el inicio y el final de la línea a dibujar. Podríamos (si fuera conveniente) sustituir los números con variables o con expresiones:

```
int x = 100;
int y = 200;
papel.drawLine(100, 100, 110, 110);
papel.drawLine(x, y, x + 50, y + 50);
papel.drawLine(x * 2, y - 8, x * 30 - 1, y * 3 + 6);
```

Las expresiones se calculan y los valores resultantes se pasan a `drawLine` para que los utilice.

He aquí otro ejemplo. El método `parseInt` requiere un parámetro de cadena de texto y el cuadro de entrada devuelve una cadena de texto. Podemos combinar éstos en una instrucción, como en el siguiente ejemplo:

```
int edad;
edad = Integer.parseInt(JOptionPane.showInputDialog(null,
 "Escriba la edad"));
```

En el ejemplo anterior no tuvimos que inventar una variable de cadena temporal para transmitir un valor de cadena de texto del cuadro de entrada a `parseInt`.

## Fundamentos de programación

- Una variable tiene un nombre que el programador puede elegir.
- Una variable tiene un tipo que el programador puede elegir.
- Una variable contiene un valor.
- El valor de una variable se puede modificar mediante una instrucción de asignación.
- Las constantes proveen valores que no cambian. Se les puede asignar un nombre significativo.

## Errores comunes de programación

- Tenga cuidado con la ortografía en los nombres de las variables. Por ejemplo, en:

```
int círculo; // error de escritura
círculo = 20;
```

la variable está mal escrita en la primera línea, ya que se utiliza un “I” (uno) en vez de una “L” minúscula. El compilador de Java se quejará de que la variable de la segunda línea no está declarada. Otro error común es utilizar un cero en vez de una “O” mayúscula.

- Es difícil detectar los errores de programación al principio. Aunque el compilador de Java nos da una indicación de la posición en la que cree que se encuentra el error, en realidad éste podría estar en una línea anterior.
- Los paréntesis deben estar balanceados; debe haber el mismo número de “(“ que de “)”.
- Al utilizar números a través de cuadros de mensaje, recuerde usar las herramientas de conversión de cadenas o unir el número a una cadena.
- Al multiplicar elementos debe colocar un \* entre ellos, mientras que en matemáticas se omite este signo. Al dividir elementos, recuerde que:
  - `int / int` nos da una respuesta `int`.
  - `double / double` nos da una respuesta `double`.
  - `int / double` y `double / int` nos dan una respuesta `double`.

## Secretos de codificación

- Para declarar variables indicamos su tipo y su nombre, como en:

```
int miVariable;
String tuVariable = "¡Saludos desde acá!";
```

- Los tipos más útiles son `int`, `double` y `String`.
- Los principales operadores aritméticos son `*`, `/`, `%`, `+` y `-`.
- El operador `+` se utiliza para unir cadenas de texto.
- Los operadores `++` y `--` se pueden utilizar para incrementar y decrementar.
- Podemos convertir números a cadenas con los métodos `Integer.toString` y `Double.toString`.
- Podemos convertir cadenas a números con los métodos `Integer.parseInt` y `Double.parseDouble`.
- Si colocamos el operador de conversión (`int`) antes de un elemento `double`, éste se convierte en un entero.
- Si colocamos el operador de conversión (`double`) antes de un elemento `int`, éste se convierte en un valor `double`.

## Nuevos elementos del lenguaje

- `int`, `double` y `string`.
- Los operadores `+`, `-`, `*`, `/`, `%`.
- `++` y `--` para incremento y decremento.
- `=` para asignación.
- `final` para las constantes.
- Conversión de tipos: las clases `Integer` y `Double`, los operadores de conversión (`double`) e (`int`).
- La clase `JOptionPane` y sus métodos `showMessageDialog` y `showInputDialog`.
- El uso de `\n` para representar una nueva línea en una cadena.

## Resumen

- Las variables se utilizan para contener (guardar) valores. Mantienen su valor hasta que éste se modifica en forma explícita (por ejemplo, mediante otra instrucción de asignación).
- Los operadores operan sobre valores.
- Una expresión es un cálculo que produce un valor. Se puede utilizar en diversas situaciones, incluyendo el lado derecho de una asignación o como parámetro para la llamada a un método.

## Ejercicios

En estos ejercicios, seleccione un nuevo nombre de clase para cada programa, pero base todos los ejercicios en el programa **CalculoÁrea**. Use cuadros de mensaje para las operaciones de salida y cuadros de entrada para las operaciones de entrada.

**4.1** Escriba un programa para calcular el volumen de una caja, dadas sus tres dimensiones.

**4.2** (a) Dado el siguiente valor:

```
double radio = 7.5;
```

utilice instrucciones de asignación para calcular la circunferencia de un círculo, el área de un círculo y el volumen de una esfera, con base en el mismo radio. Muestre los resultados con cuadros de mensaje. El mensaje debe indicar qué es el resultado, en vez de sólo mostrar un número. Use la constante **Math.PI**, como en:

```
volumen = (4 * Math.PI / 3) * radio * radio * radio;
```

**4.3** Escriba un programa que reciba como entrada tres calificaciones de exámenes representadas por enteros y muestre la calificación promedio como un valor **double**. Verifique su respuesta con una calculadora.

**4.4** Escriba un programa que reciba como entrada tres calificaciones de exámenes representadas por números **double** y muestre la calificación promedio como un valor **double**. Verifique su respuesta con una calculadora.

**4.5** Suponga que un grupo de personas tienen que pagar impuestos del 20% de sus ingresos. Obtenga el valor del ingreso por medio de un cuadro de entrada; después calcule y muestre la cantidad inicial, la cantidad después de las deducciones y la cantidad que se dedujo. Asegúrese de que los usuarios puedan entender fácilmente la salida de su programa. Modifique su programa para que utilice una constante **final** para la tasa de impuestos.

**4.6** Use tipos **int** para escribir un programa que convierta una temperatura en Fahrenheit a su equivalente en Celsius (centígrados). La fórmula es:

```
c = (f - 32) * 5 / 9
```

- 4.7** Escriba un programa que reciba como entrada un número entero de segundos; después debe convertir este valor en horas, minutos y segundos. Por ejemplo, 3669 segundos deberían mostrarse en el cuadro de mensaje como:

H:1 M:1 S:9

- 4.8** Este problema está relacionado con las resistencias eléctricas, las cuales se "resisten" al flujo de la corriente eléctrica que pasa a través de ellas. Las mangueras son una analogía: una manguera delgada tiene una alta resistencia al agua y una gruesa, una baja resistencia. Imagine que tenemos dos mangueras. Si las conectamos en serie se obtendría una mayor resistencia, y si las conectamos en paralelo se reduciría la resistencia (ya que obtendríamos el equivalente a una manguera más gruesa). Calcule y muestre la resistencia en serie con base en:

$$\text{serie} = r_1 + r_2$$

y la resistencia en paralelo, con base en:

$$\text{paralelo} = \frac{r_1 \times r_2}{r_1 + r_2}$$

Reciba como entrada los valores para  $r_1$  y  $r_2$ .

- 4.9** Suponga que necesitamos instalar cierto software en una máquina europea dispensadora de bebidas. He aquí los detalles: todos los productos cuestan menos de 1 euro (100 centavos de euro) y una moneda de 1 euro es la denominación más alta que podemos insertar. Dado el monto insertado y el costo del producto, su programa debe regresar cambio utilizando el menor número de monedas. Por ejemplo, si un producto cuesta 45 centavos y pagamos con 100 centavos, el resultado debería ser una serie de cuadros de mensaje (uno para cada moneda) de la siguiente forma:

```
La cantidad de monedas de 50 centavos es 1
La cantidad de monedas de 20 centavos es 0
La cantidad de monedas de 10 centavos es 0
La cantidad de monedas de 5 centavos es 1
La cantidad de monedas de 2 centavos es 0
La cantidad de monedas de 1 centavo es 0
```

Sugerencia: trabaje con centavos y utilice el operador % todas las veces que pueda. Las monedas de euro son:

100, 50, 20, 10, 5, 2, 1

## Respuestas a las prácticas de autoevaluación

- 4.1** `volumen` – permitido, estilo correcto.  
`ÁREA` – permitido, pero es preferible usar `área`.  
`Longitud` – permitido, pero es preferible usar la 'l' minúscula.  
`3lados` - no está permitido ya que empieza con un dígito.  
`lado1` – permitido, estilo correcto.  
`lonitud` – permitido, aun cuando está mal escrita la palabra "longitud".  
`Misalario` – permitido, pero es preferible usar `miSalario`.  
`su salario` – no está permitido (no se permiten espacios en medio de un nombre).  
`tamañoPantalla` - permitido, estilo correcto.
- 4.2** En la línea 2, `b` no está inicializada. Se producirá un error de compilación debido a que estamos tratando de almacenar una variable no asignada en `a`.
- 4.3** Los valores finales de `a`, `b`, `c` y `d` son `5`, `7`, `12` y `8`.
- 4.4** Por desgracia usted no recibe nada, ya que primero se calcula `(1 / 2)` y se obtiene un 0. Es mejor que multiplique por 0.5.
- 4.5** Los valores finales de `a`, `b`, `c` y `d` son `2`, `8`, `1` y `0`.
- 4.6**
- ```
int totalSegundos = 307;
int segundos, minutos;
minutes = totalSegundos / 60;
segundos = totalSegundos % 60;
```
- 4.7** Los cuadros de mensaje muestran las cadenas `5555` y `5510`, respectivamente.
En el primer caso procedemos de izquierda a derecha, uniendo cadenas. En el segundo caso se llevan a cabo las operaciones dentro de los paréntesis y se obtiene el entero `10`. Después se lleva a cabo la unión de cadenas.
- 4.8** Los valores finales de `m`, `n` y `s` son `334`, `37` y `64`.
- 4.9** Los valores de las variables `int i, j y k` son `2`, `3` y `3`, y los valores de las variables `double a, b y c` son `3.0`, `3.0` y `2.0`.
- 4.10**
- ```
final double cmPorInch = 2.54;
double cm, pulgadas = 23.6;
cm = pulgadas * cmPorPulg;
```

# CAPÍTULO

# 5



## Métodos y parámetros

En este capítulo conoceremos cómo:

- Escribir métodos.
- Utilizar parámetros formales y actuales.
- Pasar parámetros a los métodos.
- Usar `return` en los métodos.
- Sobrecargar métodos.

### ● Introducción

Los programas grandes pueden ser complejos, difíciles de comprender y de depurar. La técnica más importante para reducir la complejidad es dividir un programa en secciones (relativamente) aisladas. Esto nos permite enfocarnos en una sección aislada sin las distracciones del programa completo. Además, si la sección tiene un nombre, podemos “llamarla” o “invocarla” (hacer que se utilice) con sólo utilizar su nombre. En cierta forma, nos permite pensar a un nivel más alto. En Java, a dichas secciones se les conoce como métodos. En el capítulo 3 utilizamos una buena cantidad de métodos gráficos predefinidos para dibujar figuras en la pantalla.

Veamos de nuevo el método `drawRect`, que podemos llamar con cuatro parámetros de la siguiente forma:

```
papel.drawRect(10, 20, 60, 60);
```

En primer lugar, al utilizar parámetros (los elementos entre paréntesis) podemos controlar el tamaño y la posición del rectángulo. Esto asegura que `drawRect` sea lo bastante flexible como para funcionar en diversas circunstancias. Los parámetros modifican sus acciones.

En segundo lugar, cabe mencionar que si no existiera `drawRect` de todas formas podríamos producir un rectángulo mediante cuatro llamadas a `drawLine`. Agrupar las cuatro instrucciones `drawLine` dentro de un método denominado `drawRect` sería una idea sensata, ya que permite al programador pensar a un nivel más alto.

## Cómo escribir sus propios métodos

En esta sección veremos cómo crear nuestros propios métodos. Empezaremos con un pequeño ejemplo para simplificar las cosas, y después veremos un ejemplo más práctico.

La Compañía Mundial de Cajas de Cartón tiene un logotipo que consiste en tres cuadrados, uno dentro de otro, como se muestra en la figura 5.1. La empresa desea utilizar el logotipo en varias posiciones en la pantalla, como se muestra en la figura 5.2. He aquí el código para dibujar dos logotipos idénticos en las posiciones (10, 20) y (100, 100):

```
// dibuja el logotipo en la esquina superior izquierda
papel.drawRect(10, 20, 60, 60);
papel.drawRect (10, 20, 40, 40);
papel.drawRect (10, 20, 20, 20);

// dibuja el logotipo en la esquina inferior derecha
papel.drawRect (100, 100, 60, 60);
papel.drawRect (100, 100, 40, 40);
papel.drawRect (100, 100, 20, 20);
```

Observe que los cuadrados son de 20, 40 y 60 píxeles, y sus esquinas superiores izquierdas están en el mismo punto. Si analiza el código observará que en esencia se repiten las tres instrucciones para dibujar un logotipo, excepto por la posición de la esquina superior izquierda del logotipo. Vamos a agrupar esas tres instrucciones para formar un método, de manera que se pueda dibujar un logotipo con una sola instrucción.

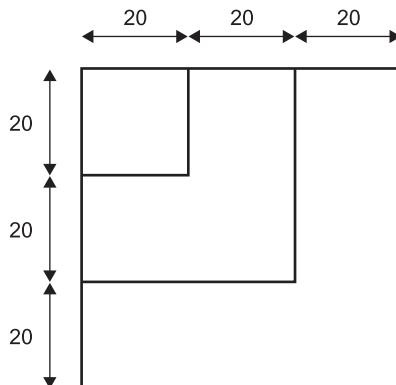
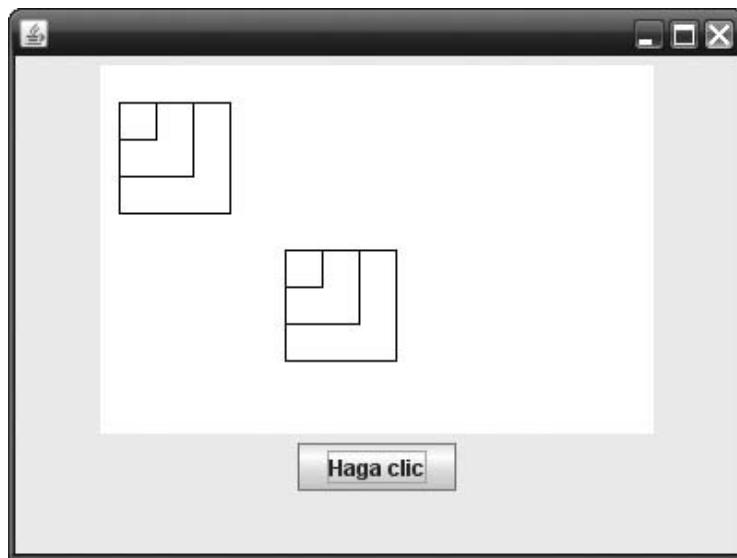


Figura 5.1 El logotipo de la empresa.



**Figura 5.2** Pantalla del programa MétodoLogo.

## Nuestro primer método

He aquí un programa completo llamado `MétodoLogo`. Este programa muestra cómo crear y utilizar un método, al cual llamaremos `dibujarLogo`. La convención de estilo de Java es empezar los nombres de los métodos con minúscula.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class MétodoLogo extends JFrame
 implements ActionListener {

 private JButton botón;
 private JPanel panel;

 public static void main(String[] args) {
 MétodoLogo marco = new MétodoLogo();
 marco.setSize(350, 300);
 marco.crearGUI();
 marco.setVisible(true);
 }
}
```

```
private void crearGUI() {
 setDefaultCloseOperation(EXIT_ON_CLOSE);
 Container ventana = getContentPane();
 ventana.setLayout(new FlowLayout());
 panel = new JPanel();
 panel.setPreferredSize(new Dimension(300, 200));
 panel.setBackground(Color.white);
 ventana.add(panel);

 botón = new JButton("Haga clic");
 ventana.add(botón);
 botón.addActionListener(this);
}

public void actionPerformed(ActionEvent event) {
 Graphics papel = panel.getGraphics();
 dibujarLogo(papel, 10, 20);
 dibujarLogo(papel, 100, 100);
}

private void dibujarLogo(Graphics áreaDibujo,
 int xPos, int yPos) {
 áreaDibujo.drawRect(xPos, yPos, 60, 60);
 áreaDibujo.drawRect(xPos, yPos, 40, 40);
 áreaDibujo.drawRect(xPos, yPos, 20, 20);
}
```



El programa tiene un panel para dibujar y un botón. La interfaz de usuario es idéntica a nuestros programas de dibujo del capítulo 3. Al hacer clic en el botón se dibujan dos logotipos, como se muestra en la figura 5.2.

El concepto de los métodos y parámetros es una importante habilidad que los programadores necesitan dominar. Ahora analizaremos el programa con detalle.

La forma general del programa es familiar: tiene instrucciones `import` en la parte superior y un botón en el que podemos hacer clic para iniciar la tarea de dibujo. La parte `actionPerformed` es la que se encarga de iniciar el dibujo. Considere el siguiente extracto:

```
private void dibujarLogo(Graphics áreaDibujo,
 int xPos, int yPos) {
```

Aquí se declara (introduce) el método; a esto se le conoce como encabezado del método. El encabezado declara el nombre del método (que nosotros tenemos la libertad de elegir) y los elementos que se deben suministrar para controlar su operación. Java utiliza los términos *parámetros actuales* y *parámetros formales* para definir estos elementos; a continuación hablaremos sobre ellos. Al resto del método se le conoce como *cuerpo* y va encerrado entre los caracteres `{` y `}`; aquí es donde se realiza el trabajo. A menudo el encabezado es una línea tan extensa que podemos dividirla en puntos adecuados (aunque no en medio de una palabra).

Una importante decisión que debe tomar el programador es el lugar desde donde se puede llamar al método. Hay dos opciones principales:

- El método sólo se puede llamar desde el interior del programa actual. En este caso utilizamos la palabra clave **private**.
- El método se puede llamar desde otro programa. En este caso utilizamos la palabra clave **public**. Los métodos como **drawRect** son ejemplos de métodos que se han declarado como **public**, puesto que son de uso general. (Para crear métodos **public** se requiere un conocimiento más profundo de los conceptos orientados a objetos; hablaremos sobre esto con más detalle en el capítulo 9).

Otras decisiones que debe tomar el programador son:

- ¿Realizará el método una tarea sin necesidad de producir un resultado? En este caso, utilizamos la palabra clave **void** después de **private**.
- ¿Calculará el método un resultado y lo devolverá a la sección de código que lo llamó (invocó)? En este caso tenemos que declarar el tipo del resultado, en vez de usar **void**. Más adelante en el capítulo veremos cómo hacerlo.

En el caso de **dibujarLogo**, su tarea es dibujar líneas en la pantalla, no proveer la respuesta a un cálculo. Por ende, utilizamos **void**.

## Cómo llamar a un método

Para llamar a un método privado en Java tiene que indicar su nombre, junto con una lista de parámetros entre paréntesis. En nuestro programa la primera llamada es:

```
dibujarLogo(papel, 10, 20);
```

Esta instrucción tiene dos efectos:

- Los valores de los parámetros se transfieren al método de manera automática. Más adelante hablaremos sobre esto con mayor detalle.
- El programa salta al cuerpo del método (las instrucciones después del encabezado) y ejecuta las instrucciones. Cuando termina con todas las instrucciones y llega al carácter **}**, la ejecución continúa en el punto desde el que se hizo la llamada al método.

Después se lleva a cabo la segunda llamada:

```
dibujarLogo(papel, 100, 100);
```

En la figura 5.3 se ilustra este proceso. Hay dos llamadas que producen dos logotipos.

## Cómo pasar parámetros

Es imprescindible comprender de la mejor forma posible cómo se transfieren (pasan) los parámetros a los métodos. En nuestro ejemplo el concepto se muestra en las siguientes líneas:

```
dibujarLogo(papel, 10, 20);

private void dibujarLogo(Graphics áreaDibujo,
 int xPos, int yPos) {
```

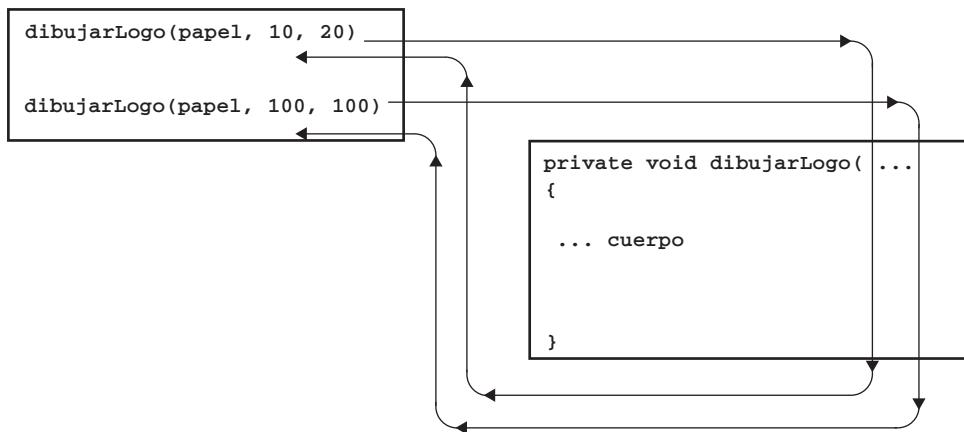


Figura 5.3 Ruta de ejecución de dos llamadas.



Figura 5.4 Proceso de transferencia de los parámetros actuales hacia los parámetros formales.

El área en la que nos debemos enfocar es en las dos listas de elementos entre paréntesis. En la llamada al método, los elementos se denominan *parámetros actuales*. En el encabezado del método, los elementos se denominan *parámetros formales*. Para aclarar esta situación, vamos a extraer los parámetros actuales y formales:

```
Parámetros actuales: papel 10 20
Parámetros formales: áreaDibujo xPos yPos
```

Recordemos la comparación que hicimos de una variable con una caja. Dentro del método hay un conjunto de cajas vacías (los parámetros) que esperan la transferencia de los valores de los parámetros. Después de la transferencia tenemos la situación que se muestra en la figura 5.4. No tenemos valores numéricos que podamos pasar al área de dibujo, por lo que nos enfocaremos en la forma en que se pasan las coordenadas.

La transferencia se realiza en un orden de izquierda a derecha. La llamada debe proporcionar el número y tipos correctos de los parámetros. Si el que hace la llamada (el usuario) envía de manera accidental los parámetros en el orden incorrecto, ¡el proceso de la transferencia no los regresará a su orden correcto! Cuando se ejecuta el método `dibujarLogo`, los valores antes mencionados controlan el proceso de dibujo. Aunque en el ejemplo anterior llamamos al método con números, podemos utilizar expresiones (es decir, incluir variables y cálculos) como en el siguiente ejemplo:

```
int x = 6;
dibujarLogo (papel, 20 + 3, 3 * 2 + 1); // 23 y 7
dibujarLogo(papel, x * 4, 20); // 24 y 20
```

## PRÁCTICAS DE AUTOEVALUACIÓN

- 5.1 ¿En qué posición se dibujarán los logotipos si se utiliza el siguiente código?

```
int a = 10;
int b = 20;
dibujarLogo(papel, a, b);
dibujarLogo(papel, b + a, b - a);
dibujarLogo(papel, b + a - 3, b + a - 4);
```

- 5.2 Podríamos reformular el método `dibujarLogo` de manera que tenga un solo parámetro: el área de dibujo. El método reformulado podría usar cuadros de entrada para que el usuario proporcione la posición de dibujo. ¿Cuál es la desventaja de usar este método?

## ● Parámetros formales y actuales

Hay dos listas entre paréntesis involucradas en nuestro estudio, por lo cual es importante aclarar el propósito de cada lista:

- El programador que escribe el código debe elegir qué elementos solicitará el método por medio de parámetros formales. Por ende, en el método `dibujarLogo` las medidas de los cuadrados anidados siempre se establecen en 20, 40 y 60, por lo que el que hace la llamada al método no necesita suministrar estos datos. Sin embargo, tal vez sí requiera variar la posición del logotipo, por lo cual hemos convertido estos elementos en parámetros.
- El escritor del método debe elegir el nombre de cada parámetro formal. Si se utilizan nombres similares en otros métodos no hay ningún problema, puesto que cada método tiene su propia copia de sus parámetros. En otras palabras, el escritor tiene la libertad de elegir cualquier nombre.
- Se debe proporcionar el tipo de cada parámetro formal y debe anteponerse al nombre del parámetro. Los tipos dependen del método en particular. Se utiliza una coma para separar un parámetro de otro. En el encabezado de `dibujarLogo` podrá ver esta disposición.
- El que hace la llamada debe proveer una lista de parámetros actuales entre paréntesis. Los parámetros deben estar en el orden que requiere el método y deben ser del tipo correcto.

Los dos beneficios de utilizar un método para dibujar el logotipo son que evitamos duplicar las tres instrucciones `drawRect` cuando se requieren varios logotipos, y que al dar un nombre a esta tarea podemos pensar a un nivel más alto.

Por último, estamos conscientes de que tal vez usted desee aplicar en otros lenguajes las habilidades de programación que aprendió aquí. Los conceptos son similares, pero la terminología es distinta: en muchos lenguajes se utiliza el término *argumento* en vez de *parámetro*. Otra de las terminologías implica el término *invocar* en vez de *llamar*.

## PRÁCTICAS DE AUTOEVALUACIÓN

**5.3** Explique cuál es el error en estas llamadas:

```
dibujarLogo(papel, 50, "10");
dibujarLogo(50, 10, papel);
dibujarLogo(papel, 10);
```

**5.4** He aquí la llamada a un método:

```
soloHazlo("Naranjas");
```

y he aquí el código del método:

```
private void soloHazlo(String fruta) {
 JOptionPane.showMessageDialog(null, fruta);
}
```

¿Qué ocurre cuando se hace una llamada a este método?

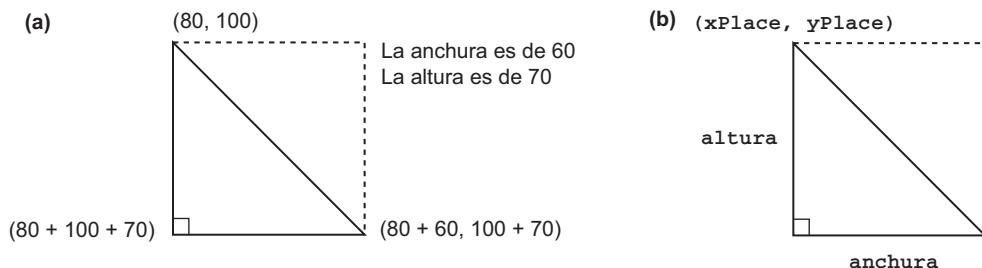
## ● Un método para dibujar triángulos

Para describir más características de los métodos debemos crear uno más útil, al cual llamaremos **dibujarTriángulo**. Como **nosotros** vamos a escribir el método (en vez de utilizar uno predefinido), podemos elegir qué tipo de triángulo se va a dibujar y los parámetros que deseamos que proporcione el que haga la llamada.

En este caso vamos a dibujar un triángulo recto orientado hacia la derecha, como se muestra en la figura 5.5(a).

Para elegir los parámetros tenemos varias posibilidades: por ejemplo, podríamos requerir que quien haga la llamada proporcione las coordenadas de las tres esquinas. Sin embargo, vamos a optar por los siguientes parámetros:

- El área de dibujo, como en el método anterior.
- Las coordenadas del punto superior del triángulo.
- La anchura del triángulo.
- La altura del triángulo.



**Figura 5.5** (a) Cálculos de las coordenadas de un triángulo. (b) Parámetros formales para **dibujarTriángulo**.

Otra manera de considerar a estas coordenadas es que especifican la posición de un rectángulo circundante para nuestro triángulo recto.

En la figura 5.5(b) se muestra un triángulo identificado con los parámetros.

Podemos dibujar las líneas en cualquier orden. Examinemos el proceso de dibujo con números primero. Como un ejemplo, dibujaremos un triángulo con la esquina superior en (80, 100), con una anchura de 60 y una altura de 70. En la figura 5.5(a) se muestran los cálculos. El proceso es el siguiente:

1. Dibujar de (80, 100) hasta (80, 100 + 70). Recuerde que la coordenada *y* se incrementa hacia abajo.
2. Dibujar de (80, 100 + 70) hasta (80 + 60, 100 + 70).
3. Dibujar de la esquina superior (80, 100) en sentido diagonal hasta (80 + 60, 100 + 70).

Asegúrese de seguir el proceso anterior; tal vez sea conveniente que lo dibuje en papel.

Observe que en nuestra explicación no simplificamos los cálculos: dejamos 100 + 70 en su forma original, en vez de usar 170. Al llegar a la codificación, la posición del triángulo y su tamaño se pasarán como parámetros separados.

He aquí un programa completo llamado `MétodoTriángulo`. Este programa contiene un método llamado `dibujarTriángulo`. También contiene el método `dibujarLogo` para ilustrar que un programa puede contener muchos métodos.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class MétodoTriángulo extends JFrame
 implements ActionListener {

 private JButton botón;
 private JPanel panel;

 public static void main(String[] args) {
 MétodoTriángulo marco = new MétodoTriángulo();
 marco.setSize(350, 300);
 marco.crearGUI();
 marco.setVisible(true);
 }

 private void crearGUI() {
 setDefaultCloseOperation(EXIT_ON_CLOSE);
 Container ventana = getContentPane();
 ventana.setLayout(new FlowLayout());

 panel = new JPanel();
 panel.setPreferredSize(new Dimension(300, 200));
 panel.setBackground(Color.white);
 ventana.add(panel);
 }
}
```

```
botón = new JButton("Haga clic");
ventana.add(botón);
botón.addActionListener(this);
}

public void actionPerformed(ActionEvent event) {
 Graphics papel = panel.getGraphics();
 dibujarLogo(papel, 10, 20);
 dibujarLogo(papel, 100, 100);
 dibujarTriángulo(papel, 100, 10, 40, 40);
 dibujarTriángulo(papel, 10, 100, 20, 60);
}

private void dibujarLogo(Graphics áreaDibujo,
 int xPos, int yPos) {
 áreaDibujo.drawRect(xPos, yPos, 60, 60);
 áreaDibujo.drawRect(xPos, yPos, 40, 40);
 áreaDibujo.drawRect(xPos, yPos, 20, 20);
}

private void dibujarTriángulo(Graphics áreaDibujo,
 int lugarX,
 int lugarY,
 int anchura,
 int altura) {
 áreaDibujo.drawLine(lugarX, lugarY,
 lugarX, lugarY + altura);
 áreaDibujo.drawLine(lugarX, lugarY + altura,
 lugarX + anchura, lugarY + altura);
 áreaDibujo.drawLine(lugarX, lugarY,
 lugarX + anchura, lugarY + altura);
}
}
```



El programa tiene un botón y un panel para dibujar. Haga clic en el botón y se dibujarán dos logotipos y dos triángulos. En la figura 5.6 se muestra el resultado.

Veamos algunos detalles sobre la codificación del método `dibujarTriángulo`:

- Decidimos llamarlo `dibujarTriángulo`, pero podemos elegir cualquier otro nombre. Pudimos haber elegido `Triángulo` o incluso `dibujarCosa`, pero `dibujarTriángulo` se ajusta a los nombres de los métodos de la biblioteca.
- Nosotros elegimos los nombres de los parámetros `áreaDibujo`, `lugarX`, `lugarY`, `anchura` y `altura`.
- El orden de los parámetros también está bajo nuestro control. Podríamos volver a codificar el método para requerir la altura antes que la anchura si quisieramos (pusimos la anchura primero debido a que muchos de los métodos de la biblioteca de Java siguen este orden).

Así que ahora tenemos nuestro triángulo. Lo utilizaremos para analizar las variables locales y también para mostrar cómo puede ser la base para métodos más poderosos.

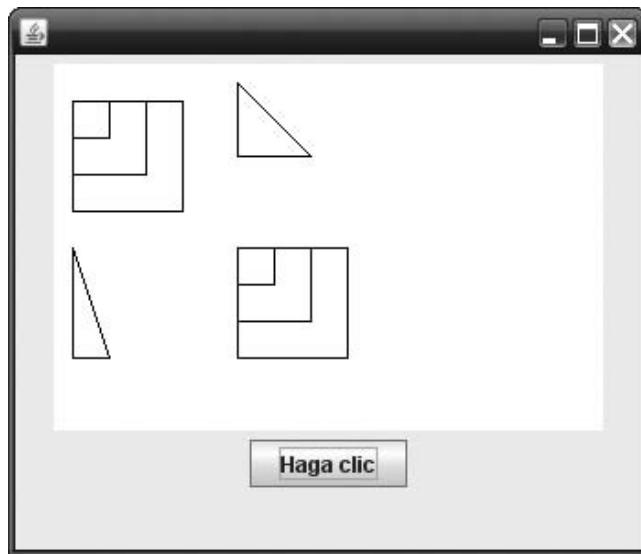


Figura 5.6 Pantalla del programa `MétodoTriángulo`.

## Variables locales

Dé un vistazo a la siguiente versión modificada de `dibujarTriángulo`, a la cual hemos llamado `dibujarTriángulo2`:

```
private void dibujarTriángulo2(Graphics áreaDibujo,
 int lugarX,
 int lugarY,
 int anchura,
 int altura) {
 int esquinaDerechaX, esquinaDerechaY;
 esquinaDerechaX = lugarX + anchura;
 esquinaDerechaY = lugarY + altura;
 áreaDibujo.drawLine(lugarX, lugarY,
 lugarX, esquinaDerechaY);
 áreaDibujo.drawLine(lugarX, esquinaDerechaY,
 esquinaDerechaX, esquinaDerechaY);
 áreaDibujo.drawLine(lugarX, lugarY,
 esquinaDerechaX, esquinaDerechaY);
}
```

La llamada a este método se hace de la misma forma que con `dibujarTriángulo`, pero en su interior utiliza dos variables llamadas `esquinaDerechaX` y `esquinaDerechaY`, las cuales se introdujeron para simplificar los cálculos. Vea la forma en que se utilizan para hacer referencia al punto del triángulo que está más a la derecha. Estas variables sólo existen dentro de `dibujarTriángulo2`.

Son locales para el método (en la terminología se dice que tienen *alcance* local). Si existen variables del mismo nombre dentro de otros métodos, entonces no hay conflicto ya que cada método utiliza su propia copia. Otra forma de ver esto es que cuando los programadores crean métodos, pueden inventar las variables locales sin tener que revisar los nombres de las variables de los demás métodos.

La función que desempeñan las variables locales es ayudar al método a realizar su trabajo, sin importar lo que haga. Las variables tienen un alcance limitado, ya que están restringidas a su propio método. Su existencia es temporal: se crean al momento de llamar al método y se destruyen cuando el método termina de ejecutarse.

## ● Conflictos de nombres

En Java, el creador de un método tiene la libertad de elegir nombres apropiados para las variables locales y los parámetros. Pero, ¿qué ocurre si se eligen nombres que estén en conflicto con otras variables? Podríamos tener lo siguiente:

```
private void metodoUno(int x, int y) {
 int z = 0;
 // código...
}

private void metodoDos(int z, int x) {
 int w = 1;
 // código...
}
```

Suponga que dos personas escribieron estos métodos. `metodoUno` tiene los parámetros `x` e `y`; además declara una variable tipo entero llamada `z`. Estos tres elementos son locales para `metodoUno`. En `metodoDos` el programador ejerce su derecho a nombrar los elementos locales y opta por usar `z`, `x` y `w`. El conflicto de nombres por la `x` (y la `z`) no representa un problema, ya que Java considera que la `x` de `metodoUno` y la `x` de `metodoDos` son distintas.

## PRÁCTICA DE AUTOEVALUACIÓN

5.5 Considere la siguiente llamada a un método:

```
int a = 3;
int b = 8;
hacerAlgo(a, b);
 JOptionPane.showMessageDialog(null, Integer.toString(a));
```

y he aquí el método en sí:

```
private void hacerAlgo(int x, int y) {
 int a = 0;
 a = x + y;
}
```

¿Qué se muestra en el cuadro de mensaje?

Vamos a ver un resumen de las herramientas para trabajar con métodos que hemos visto hasta ahora (más adelante incluiremos la instrucción `return`).

- La forma general de la declaración de un método que no produce un resultado y recibe los parámetros por valor es:

```
private void unNombre(lista de parámetros formales) {
 cuerpo
}
```

El programador elige el nombre del método.

- La lista de parámetros formales es una lista de tipos y nombres. Si un método no necesita parámetros, utilizamos paréntesis vacíos para la lista de parámetros al momento de declararlo, y utilizamos paréntesis vacíos para la lista de parámetros actuales al momento de llamarlo.

```
private void miMétodo() {
 cuerpo
}
```

y la llamada al método es:

```
miMétodo();
```

- Una clase puede contener cualquier cantidad de métodos, en cualquier orden. En este capítulo nuestros programas sólo contienen una clase. La esencia de su distribución es:

```
public class UnaClase... {
 public static void main(lista de parámetros...) {
 cuerpo
 }

 private void unNombre(lista de parámetros...) {
 cuerpo
 }

 private void otroNombre(lista de parámetros...) {
 cuerpo
 }
}
```

En el capítulo 9 veremos cómo usar las palabras clave `public` y `class`. Por ahora sólo basta con tener en cuenta que una clase puede agrupar a una serie de métodos.

## Métodos para manejar eventos y main

Una clase contiene un conjunto de métodos. Algunos de ellos los escribimos nosotros (como `dibujarLogo`) y los llamamos de manera explícita. Sin embargo, hay otros métodos que creamos pero no llamamos, como `main` y `actionPerformed`.

Si nunca llamáramos a un método, no tendría efecto. Sin embargo, los anteriores métodos sí son llamados, sólo que desde el sistema en tiempo de ejecución de Java en vez de llamarlos en forma explícita desde nuestro programa.

- Cuando un programa empieza a ejecutarse, se hace una llamada de manera automática al método `main` antes de que ocurra cualquier otra cosa. Su tarea principal es llamar a algunos métodos que crean la GUI mediante la adición (por ejemplo) de botones a la ventana.
- El sistema en tiempo de ejecución de Java llamará al método `actionPerformed` cada vez que se haga clic en un botón. Este proceso no es totalmente automático: el programa tiene que declarar que va a “escuchar” o estar atento a los clics de los botones. En el capítulo 6 hablaremos sobre esto.

## ● La instrucción `return` y los resultados

En nuestros ejemplos anteriores de parámetros formales y actuales pasamos valores hacia los métodos para que éstos los utilicen. Sin embargo, con frecuencia es necesario codificar métodos que realicen un cálculo y envíen un resultado al resto del programa, de manera que este resultado se pueda utilizar en cálculos posteriores. En este caso podemos utilizar la instrucción `return`. Veamos a continuación un método que calcula el área de un rectángulo, dados sus dos lados como parámetros de entrada. He aquí un programa completo llamado `MétodoÁrea`, el cual muestra al método y una llamada:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class MétodoÁrea extends JFrame
 implements ActionListener {

 private JButton botón;
 private JPanel panel;

 public static void main(String[] args) {
 MétodoÁrea marco = new MétodoÁrea();
 marco.setSize(400, 300);
 marco.crearGUI();
 marco.setVisible(true);
 }

 private void crearGUI() {
 setDefaultCloseOperation(EXIT_ON_CLOSE);
 Container ventana = getContentPane();
 ventana.setLayout(new FlowLayout());

 panel = new JPanel();
 panel.setPreferredSize(new Dimension(300, 200));
 panel.setBackground(Color.white);
 ventana.add(panel);

 botón = new JButton("Haga clic");
 ventana.add(botón);
 botón.addActionListener(this);
 }
}
```

```

public void actionPerformed(ActionEvent event) {
 int a;
 a = áreaRectángulo(10, 20);
 JOptionPane.showMessageDialog(null, "El área es: " + a);
}

private int áreaRectángulo(int longitud, int anchura) {
 int área;
 área = longitud * anchura;
 return área;
}

```



Hay varias características nuevas en este ejemplo que van de la mano.

Consideré el encabezado del método:

```
private int áreaRectángulo(int longitud, int anchura) {
```

En vez de **void** especificamos el tipo de elemento que el método debe regresar al que hizo la llamada. En este caso, y como estamos multiplicando dos valores **int**, la respuesta también es de tipo **int**.

La elección del tipo de este elemento depende del problema. Por ejemplo, podría ser entero o cadena, pero también podría ser un objeto más complicado, tal como un botón. El programador que escribe el método elige el tipo de valor que se va a devolver.

Para devolver un valor del método utilizamos la instrucción **return** de la siguiente manera:

```
return expresión;
```

La expresión podría ser un número, una variable o un cálculo (o incluso la llamada a un método), pero debe ser del tipo correcto, según lo especificado en la declaración del método (su encabezado). Además, la instrucción **return** hace que el método actual deje de ejecutarse y regresa de inmediato al lugar en el que se encontraba dentro del método que hizo la llamada. Ahora veamos cómo se puede llamar a un método que devuelve un resultado.

La siguiente es una manera de cómo **no** llamar a dicho método. No se deben utilizar como instrucciones completas; por ejemplo:

```
áreaRectángulo(10, 20); // no
```

El programador debe asegurarse de “consumir” o “utilizar” el valor devuelto. He aquí una metodología para comprender cómo devolver valores: imagine que se elimina la llamada al método (el nombre y la lista de parámetros) y se sustituye por el resultado devuelto. Si el código resultante tiene sentido, entonces Java le permitirá realizar dicha llamada. Vea el siguiente ejemplo:

```
respuesta = áreaRectángulo(30, 40);
```

El resultado es 1200, y si sustituimos la llamada por este resultado, obtenemos lo siguiente:

```
respuesta = 1200;
```

Esto es código válido en Java, pero si utilizamos:

```
áreaRectángulo(30, 40);
```

al sustituir el resultado devuelto se produciría una instrucción de Java que sólo tendría un número:

```
1200;
```

lo cual no tiene significado (aunque en el sentido estricto, el compilador de Java permitirá que se ejecute la llamada anterior a `áreaRectángulo`. Sin embargo, no tiene caso ignorar el resultado devuelto de un método, cuyo propósito principal es precisamente devolver dicho resultado).

He aquí algunas otras formas válidas en las que podríamos consumir el resultado:

```
int n;
n = áreaRectángulo(10, 20);
JOptionPane.showMessageDialog(null, "el área es de " +
 áreaRectángulo(3, 4));
n = áreaRectángulo(10, 20) * áreaRectángulo(7, 8);
```

## PRÁCTICA DE AUTOEVALUACIÓN

- 5.6 Utilice lápiz y papel para trabajar con las instrucciones anteriores; sustituya los resultados por las llamadas.

Para completar nuestra explicación sobre `return` cabe mencionar que se puede utilizar con métodos `void`. En este caso, debemos utilizar `return` sin especificar un resultado, como en el siguiente ejemplo:

```
private void demo(int n) {
 // hacer algo
 return;
 // hacer otra cosa
}
```

Esto se puede utilizar cuando queremos que el método termine en una instrucción que no sea la última.

Ahora veamos una manera alternativa de codificar nuestro ejemplo del área:

```
private int áreaRectángulo2(int longitud, int anchura) {
 return longitud * anchura;
}
```

Debido a que podemos utilizar `return` con expresiones, hemos omitido la variable `área` en `áreaRectángulo2`.

Dichas reducciones en el tamaño del programa no siempre son beneficiosas, ya que al reducir los nombres representativos se puede disminuir la claridad, lo que por ende puede provocar que se requiera más tiempo de depuración y prueba.

## PRÁCTICA DE AUTOEVALUACIÓN

**5.7** El siguiente método se llama `doble` y devuelve el doble del valor de su parámetro `int`:

```
private int doble(int n) {
 return 2 * n;
}
```

Dadas las siguientes llamadas al método:

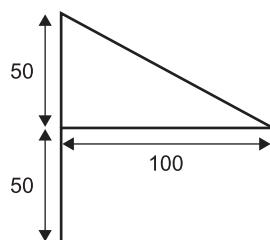
```
int n = 3;
int r;
r = doble(n);
r = doble(n + 1);
r = doble(n) + 1;
r = doble(3 + 2 * n);
r = doble(doble(n));
r = doble(doble(n + 1));
r = doble(doble(n) + 1);
r = doble(doble(doble(n))));
```

Indique el valor devuelto para cada llamada.

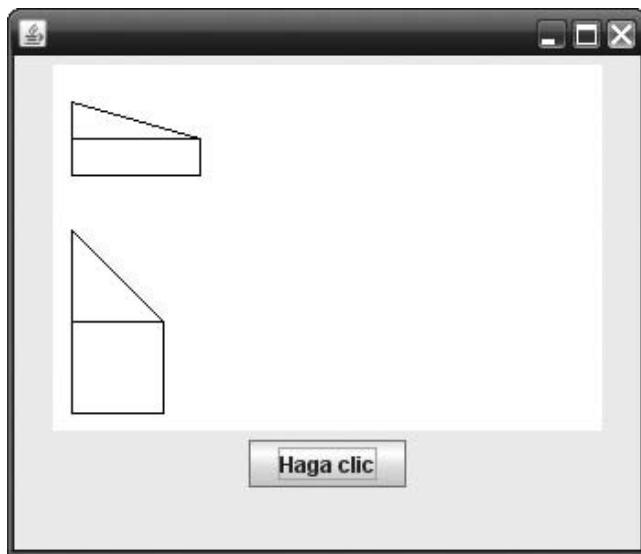
### Cómo construir métodos a partir de otros métodos: dibujarCasa

Como ejemplo de métodos que utilizan otros métodos, vamos a crear un método que dibuja una casa sencilla con una sección transversal, la cual se muestra en la figura 5.7. La altura del techo es la misma que la altura de las paredes, y la anchura de las paredes es la misma que la anchura del techo. Vamos a utilizar los siguientes parámetros `int`:

- La posición horizontal del punto superior derecho del techo.
- La posición vertical del punto superior derecho del techo.



**Figura 5.7** Una casa cuya anchura es de 100 y la altura del techo es de 50.



**Figura 5.8** Pantalla del programa `DemoCasa`.

- La altura del techo (excluyendo la pared).
- La anchura de la casa. El triángulo para el techo y el rectángulo para las paredes tienen la misma anchura.

Utilizaremos el método `drawRect` de la biblioteca de Java y utilizaremos nuestro propio método `dibujarTriángulo`.

He aquí el nuevo código. La configuración de la interfaz de usuario es idéntica a la de nuestro programa anterior. La imagen resultante se muestra en la figura 5.8.

```
public void actionPerformed(ActionEvent event) {
 Graphics papel = panel.getGraphics();
 dibujarCasa(papel, 10, 20, 70, 20);
 dibujarCasa(papel, 10, 90, 50, 50);
}

private void dibujarCasa(Graphics áreaDibujo,
 int techoSupX,
 int techoSupY,
 int anchura,
 int altura) {
 dibujarTriángulo(áreaDibujo, techoSupX, techoSupY, anchura, altura);
 áreaDibujo.drawRect(techoSupX,
 techoSupY + altura, anchura, altura);
}
```

```
private void dibujarTriángulo(Graphics áreaDibujo,
 int lugarX,
 int lugarY,
 int anchura,
 int altura) {
 áreaDibujo.drawLine(lugarX, lugarY,
 lugarX, lugarY + altura);
 áreaDibujo.drawLine(lugarX, lugarY + altura,
 lugarX + anchura, lugarY + altura);
 áreaDibujo.drawLine(lugarX, lugarY, lugarX + anchura,
 lugarY + altura);
}
```

El programa es bastante claro si consideramos que:

- Los métodos regresan al punto desde el cual se llamaron, por lo cual la secuencia de llamadas es:
  - `actionPerformed` llama a `dibujarCasa`.
  - `dibujarCasa` llama a `drawRect`.
  - `dibujarCasa` llama a `dibujarTriángulo`.
  - `dibujarTriángulo` llama a `drawLine` (tres veces).
- Los parámetros pueden ser expresiones, por lo que se evalúa `lugarY + altura` y después se pasa el resultado a `drawLine`.
- Las variables `anchura` y `altura` de `dibujarCasa` y las variables `anchura` y `altura` de `dibujarTriángulo` son totalmente distintas. Sus valores se almacenan en diferentes lugares.

Aquí podemos ver que lo que hubiera podido ser un programa más grande se ha escrito como un programa corto, dividido en métodos y con nombres representativos. Esto ilustra el poder que se obtiene al utilizar métodos.

## Cómo construir métodos a partir de otros métodos: `áreaCasa`

Ahora veamos un método que calcula y devuelve el área de la sección transversal de nuestra casa. Vamos a utilizar nuestro método `áreaRectángulo` existente y escribiremos un método `áreaTriángulo` basado en lo siguiente:

```
área = (base * altura) / 2;
```

He aquí una llamada a `áreaCasa`:

```
int área = áreaCasa(10, 10);
JOptionPane.showMessageDialog(null,
 "El área de la casa es de " + área);
```

y he aquí los métodos que utiliza esta llamada:

```
private int áreaCasa(int anchura, int altura) {
 return áreaRectángulo(anchura, altura) +
 áreaTriángulo(anchura, altura);
}

private int áreaRectángulo(int longitud, int anchura) {
 int área;
 área = longitud * anchura;
 return área;
}

private int áreaTriángulo(int base, int altura) {
 return (base*altura)/2;
}
```

Recuerde que en nuestra casa simplificada, la altura de la parte del techo es igual a la altura de la pared, por lo que sólo necesitamos pasar la anchura y la altura. He aquí cómo funcionan los métodos en conjunto:

- `áreaCasa` llama a `áreaRectángulo` y a `áreaTriángulo`.
- `áreaCasa` suma los dos resultados que se devuelven.
- `áreaCasa` devuelve el resultado final.

Cabe mencionar que `áreaTriángulo` obtiene sus entradas por medio de parámetros en vez de solicitarlas al usuario por medio de cuadros de entrada. Esto lo hace flexible. Por ejemplo, en este caso los valores de sus parámetros se crean en `áreaCasa`.

Hasta ahora, en nuestro estudio sobre los métodos hemos visto lo siguiente:

- Pasar parámetros a un método.
- Pasar un valor fuera de un método mediante `return`.

## ● La palabra clave `this` y los objetos

Probablemente esté leyendo este libro debido a que Java es un lenguaje orientado a objetos, pero tal vez se esté preguntando por qué no hemos mencionado a los objetos en este capítulo. La verdad es que los métodos y los objetos tienen una conexión vital. Al ejecutar programas pequeños en Java, estamos ejecutando una instancia de una clase; es decir, un objeto. Este objeto contiene métodos (como `dibujarLogo`).

Cuando un objeto llama a un método que está declarado en su interior, podemos simplificar la llamada utilizando:

```
dibujarLogo(papel, 50, 10);
```

- podemos utilizar la notación de objetos completa, como en el siguiente ejemplo:

```
this.dibujarLogo(papel, 50, 10);
```

`this` es una palabra clave de Java y representa al objeto actual en ejecución. Por lo tanto, ha estado utilizando la programación orientada a objetos sin darse cuenta de ello. He aquí algunos ejemplos:

```
// funciona como se esperaba
papel.drawLine(10, 10, 100, 100);

// error de compilación
this.drawLine(10, 10, 100, 100);
```

En el ejemplo anterior se detectará un error, ya que estamos pidiendo a Java que localice el método `drawLine` dentro del objeto actual. De hecho, `drawLine` existe fuera del programa en la clase `Graphics`.

## Sobrecarga de métodos

Nuestro método `áreaTriángulo` es útil en cuanto a que puede trabajar con parámetros actuales que tengan cualquier nombre. La desventaja es que deben ser enteros. Pero ¿qué tal si quisieramos trabajar con dos variables `double`? Podríamos codificar *otro* método:

```
private double áreaTriángulodouble(double base, double altura) {
 return 0.5 * (base * altura);
}
```

Sin embargo, sería conveniente utilizar el mismo nombre para ambos métodos; en Java podemos hacerlo. He aquí cómo codificaríamos las declaraciones de estos métodos:

```
private int áreaTriángulo(int base, int altura) {
 return (base * altura) / 2;
}

private double áreaTriángulo(double base, double altura) {
 return (base * altura) * 0.5;
}

private double áreaTriángulo(double lado1, double lado2,
 double angulo) {
 return 0.5 * lado1 * lado2 * Math.sin(angulo);
}
```

Junto a las versiones `int` y `double` colocamos una versión adicional, para aquellos que estén familiarizados con la trigonometría. Aquí el área se calcula con base en la longitud de dos lados y el seno del ángulo entre ellos, en radianes (`Math.sin` provee la función seno). Esta versión de `áreaTriángulo` tiene tres parámetros `double`. Aquí llamamos a los tres métodos y mostramos los resultados en cuadros de mensaje:

```
public void actionPerformed(ActionEvent event) {
 double da = 9.5, db = 21.5;
 int ia = 10, ib = 20;
 double angulo = 0.7;
 JOptionPane.showMessageDialog(null,
 "El área del triángulo es de "+áreaTriángulo(ia, ib));
 JOptionPane.showMessageDialog(null,
 "El área del triángulo es de "+áreaTriángulo(da, db));
 JOptionPane.showMessageDialog(null,
 "El área del triángulo es de "+áreaTriángulo(da, db, angulo));
}
}
```

¿Cómo decide Java cuál método utilizar? Hay tres métodos llamados **áreaTriángulo**, por lo que Java busca además del nombre el número de parámetros actuales en la llamada y sus tipos. Por ejemplo, si llamamos a **áreaTriángulo** con dos parámetros **double**, busca la declaración apropiada de **áreaTriángulo** con dos (y sólo dos) parámetros **double**. El código que contienen los métodos puede ser diferente; es el número de parámetros y sus tipos lo que determina a cuál método llamar. A la combinación del número de parámetros y sus tipos se le conoce como la *firma* del método.

Si el método devuelve un resultado, el tipo de este valor de retorno no participa a la hora de determinar cuál método se va a llamar; es decir, son los tipos de los parámetros del método los que deben ser distintos.

Lo que hemos hecho aquí se denomina *sobre cargar* un método. Hemos sobre cargado el método **áreaTriángulo** con varias posibilidades.

Por lo tanto, si usted escribe métodos que realicen tareas similares pero tengan distintos números y/o tipos de parámetros, es conveniente utilizar la sobre carga y elegir el **mismo** nombre en vez de inventar un nombre artificialmente distinto.

## PRÁCTICA DE AUTOEVALUACIÓN

- 5.8 Escriba dos métodos, ambos llamados **sumarNúmeros**. Uno debe sumar dos enteros y devolver su suma. El otro debe sumar tres enteros y devolver la suma. Provea ejemplos de cómo llamar a sus dos métodos.

## Fundamentos de programación

- Un método es una sección de código que tiene asignado un nombre. Para llamar al método usamos su nombre.
- Podemos codificar métodos **void** o métodos que devuelvan un resultado.
- Podemos pasar parámetros a un método.
- Si podemos identificar una tarea bien definida en nuestro código, podemos separarla y escribirla como un método.

## Errores comunes de programación

- El encabezado del método debe incluir los nombres de los tipos. El siguiente código está mal:

```
private void métodoUno(x){ // incorrecto
```

Debemos utilizar algo como esto:

```
private void métodoUno(int x) {
```

- La llamada a un método no debe incluir los nombres de los tipos. Por ejemplo, en vez de:

```
métodoUno(int y); //
```

debemos usar:

```
métodoUno(y);
```

- Al llamar a un método debemos suministrar el número correcto de parámetros y los tipos correctos de éstos.
- Siempre debemos consumir un valor devuelto de alguna forma. El siguiente estilo de llamada no consume un valor devuelto:

```
unMétodo(e, f); //
```

## Secretos de codificación

- El patrón general para los métodos toma dos formas. En primer lugar, para un método que no devuelve un resultado, la forma de declararlo es:

```
private void nombreMétodo(lista de parámetros formales) {
 ... cuerpo
}
```

y debemos invocar el método mediante una instrucción, como en el siguiente ejemplo:

```
nombreMétodo(lista de parámetros actuales);
```

- Para un método que devuelve un resultado, la forma de declararlo es:

```
private tipo nombreMétodo(lista de parámetros actuales) {
 ... cuerpo
}
```

Se puede especificar cualquier tipo o clase para el valor devuelto. Podemos invocar el método como parte de una expresión, como en el siguiente ejemplo:

```
n = nombreMétodo(a, b);
```

- El cuerpo de un método que devuelve un resultado debe incluir una instrucción `return` con el tipo de valor correcto.
- Cuando un método no tiene parámetros debemos usar paréntesis vacíos () tanto en la declaración como en la llamada.
- El escritor del método crea la lista de parámetros formales. Cada parámetro necesita un nombre y un tipo.
- El que hace la llamada al método escribe la lista de parámetros actuales. Esta lista consta de una serie de elementos en el orden correcto y con los tipos correctos. A diferencia de los parámetros dentro del encabezado de un método, aquí no se especifican los nombres de los tipos.

## Nuevos elementos del lenguaje

- La declaración de métodos privados.
- La llamada (o invocación) a un método, que consta del nombre del método y sus parámetros.
- El uso de `return` para salir de un método que no sea `void` y devolver al mismo tiempo un valor.
- El uso de `return` para salir de un método `void`.
- El uso de la sobrecarga.
- El uso de `this` para representar al objeto actual.

## Resumen

- Los métodos contienen subtareas de un programa.
- Podemos pasar parámetros a los métodos.
- Utilizar un método es algo conocido como *llamar* (o *invocar*) al método.
- Los métodos que no son `void` devuelven un resultado.

## Ejercicios

Para los ejercicios que dibujan figuras, base su código en el programa **MétodoLogo**. Para los programas que sólo necesitan cuadros de mensaje y entrada, base su código en el programa **ÁreaRectángulo**.

El primer grupo de problemas sólo requiere de métodos **void**.

- 5.1 Escriba un método llamado **mostrarNombre** con un parámetro de cadena. Debe mostrar el nombre suministrado en un cuadro de mensaje. Para probar el programa, introduzca un nombre mediante un cuadro de entrada y después llame a **mostrarNombre**.
- 5.2 Escriba un método llamado **mostrarNombres** con dos parámetros de cadena que representen su primer nombre y su apellido paterno. El método debe mostrar su primer nombre en un cuadro de mensaje y después mostrar su apellido paterno en otro cuadro de mensaje.
- 5.3 Escriba un método llamado **mostrarIngresos** con dos parámetros enteros que representen el salario de un empleado y el número de años trabajados. El método debe mostrar el total de ingresos obtenidos por el empleado en un cuadro de mensaje, suponiendo que haya obtenido la misma cantidad de ingresos cada año. El programa deberá obtener los valores mediante cuadros de entrada antes de llamar a **mostrarIngresos**.
- 5.4 Codifique un método que dibuje un círculo, dadas las coordenadas de su centro y su radio. Su encabezado deberá ser el siguiente:

```
private void dibujarCírculo(Graphics áreaDibujo,
 int xCentro, int yCentro, int radio)
```

- 5.5 Codifique un método llamado **dibujarCalle** que dibuje una calle llena de casas, para lo cual debe utilizar el método **dibujarCasa** que utilizamos en este capítulo. Para los fines de este ejercicio una calle contiene cuatro casas y debe haber un espacio de 20 píxeles entre cada casa. Los parámetros deben proporcionar la ubicación y el tamaño de la casa de más a la izquierda, y deben ser idénticos a los de **dibujarCasa**.
- 5.6 Codifique un método que se llame **dibujarCalleEnPerspectiva** y tenga los mismos parámetros que el método del ejercicio 5.5. Sin embargo, cada casa debe ser un 20% más pequeña que la casa a su izquierda.

Los siguientes programas requieren métodos que devuelvan un resultado.

- 5.7 Escriba un método que devuelva el equivalente en pulgadas de su parámetro en centímetros. La siguiente es una llamada de ejemplo:

```
double pulgadas = equivalentePulgadas(10.5);
```

Multiplique los centímetros por **0.394** para calcular las pulgadas.

- 5.8 Escriba un método que devuelva el volumen de un cubo, dada la longitud de uno de sus lados. La siguiente es una llamada de ejemplo:

```
double vol = volumenCubo(1.2);
```

- 5.9 Escriba un método que devuelva el área de un círculo, dado su radio como parámetro. La siguiente es una llamada de ejemplo:

```
double a = áreaCírculo(1.25);
```

El área de un círculo se obtiene en base a la fórmula **Math.PI \* r \* r**. Aunque podríamos utilizar un número como **3.14**, **Math.PI** nos proporciona un valor más preciso.

- 5.10** Escriba un método llamado **segseEn** que acepte tres enteros, los cuales representarán el tiempo en horas, minutos y segundos. Debe devolver el tiempo total en segundos. La siguiente es una llamada de ejemplo:

```
int totalSegs = segsEn(1, 1, 2); // devuelve 3662
```

- 5.11** Escriba un método que devuelva el área de un cilindro sólido. Decida qué parámetros utilizar. Debe llamar al método **áreaCírculo** del ejercicio 5.9 para que le ayude a calcular el área de las partes superior e inferior (la circunferencia de un círculo se obtiene mediante la fórmula  $2 * \text{Math.PI} * r$ ).
- 5.12** Escriba un método llamado **incremento** que sume 1 a su parámetro entero. La siguiente es una llamada de ejemplo:

```
int n = 3;
int a = incremento(n); // devuelve 4
```

- 5.13** Escriba un método llamado **resultadoCincoAnios** con dos parámetros:

- Una cantidad **double** que se invierte al principio.
- Una tasa de interés **double** (por ejemplo, **1.5** especifica el 1.5% de interés anual).

El método debe devolver la cantidad después de cinco años de interés compuesto.

Sugerencia: después de un año, la nueva cantidad es:

```
cantidad = cantidad * (1 + interes / 100);
```

- 5.14** Escriba un método llamado **diferenciaDeTiempoEnSegs** con seis parámetros y que devuelva un resultado entero. Debe recibir dos tiempos en horas, minutos y segundos; además debe devolver la diferencia entre esos dos tiempos en segundos. Para resolver este problema llame al método **segseEn** (ejercicio 5.10) desde el interior de su método **diferenciaDeTiempoEnSegs**.

Los siguientes problemas son acerca de la sobrecarga de métodos:

- 5.15** Utilice cualquier programa que contenga el método **segseEn**. Agregue un método que también se llame **segseEn** y que tenga sólo dos argumentos, uno para los minutos y otro para los segundos.
- 5.16** Localice el método **dibujarCírculo** que escribió en el ejercicio 5.4. Agregue otro método **dibujarCírculo** a su programa **dibujarCírculo** con los siguientes parámetros:

- Un área de dibujo.
- La posición **x** e **y** del centro.

Se debe dibujar un círculo con un radio de **50** en el punto especificado.

## Respuestas a las prácticas de autoevaluación

- 5.1** En (10, 20), (30, 10) y (27, 26).
- 5.2** Ahora el método es más inflexible. Los cuadros de entrada aparecerían cada vez que llamáramos al método. En la versión original del método, el que hace la llamada puede obtener valores para la posición en una variedad de formas antes de pasarlos como parámetros.
- 5.3** En la primera llamada no se deben utilizar las comillas, ya que indican una cadena y no un entero.

En la segunda llamada el orden debe ser:

```
papel, 50, 10
```

En la tercera llamada falta un parámetro.

- 5.4** Aparecerá un cuadro de mensaje mostrando **Naranjas**.
- 5.5** El cuadro de mensaje muestra el valor original de **a**, que es **3**. La **a** que se convierte en **0** dentro del método es una variable local.
- 5.6** He aquí las etapas para sustituir una llamada por su resultado. Para:

```
n = áreaRectángulo(10, 20);
```

tenemos:

```
n = 200;
```

Para la línea:

```
JOptionPane.showMessageDialog(null, "el área es de " +
áreaRectángulo(3, 4));
```

tenemos las siguientes etapas:

```
JOptionPane.showMessageDialog(null, "el área es de " + 12);
JOptionPane.showMessageDialog(null, "el área es de 12");
```

Para la línea:

```
n = áreaRectángulo(10, 20) * áreaRectángulo(7, 8);
```

tenemos las etapas:

```
n = 200 * 56;
n = 11200;
```

- 5.7** Los valores que recibe **r** son:

```
6
8
7
18
12
16
14
24
```

```
5.8 private int sumarNúmeros(int a, int b) {
 return a + b;
 }

 private int sumarNúmeros(int a, int b, int c) {
 return a + b + c;
 }
```

Puede llamar a los métodos anteriores así:

```
public void actionPerformed(ActionEvent event) {
 int suma2, suma3;
 int x = 22, y = 87, z = 42;
 suma2 = sumarNúmeros(x, y);
 suma3 = sumarNúmeros(x, y, z);
}
```

## CAPÍTULO

# 6



# Cómo usar objetos

En este capítulo conoceremos cómo:

- Usar las variables de instancia privadas.
- Usar las clases de biblioteca.
- Usar `new` y los constructores.
- Manejar eventos.
- Usar la clase `Random`.
- Usar las clases etiqueta, campo de texto, panel y botón de Swing.
- Usar las clases control deslizante, temporizador e ícono de imagen de Swing.

### ● Introducción

En este capítulo veremos los objetos a un nivel más detallado. En especial analizaremos el uso de las clases de las bibliotecas de Java. Debe tener en cuenta que, aunque hay muchos cientos de estos objetos, los principios para usarlos son similares.

He aquí una analogía: para leer un libro (cualquiera que sea) hay que abrirlo por su parte frontal, leer una página y después avanzar a la siguiente página. Sabemos qué hacer con un libro. Lo mismo pasa con los objetos. Después de usar unos cuantos de ellos sabemos qué buscar cuando se nos presenta uno nuevo.

## ● Variables de instancia

Para lidiar con problemas más avanzados necesitamos introducir un nuevo lugar en donde podamos declarar variables. Hasta ahora hemos utilizado las palabras clave `int` y `double` para declarar variables locales dentro de los métodos. Pero las variables locales por sí solas no pueden lidiar con la mayoría de los problemas.

A continuación veremos un programa (`ContadorAutos`, figura 6.1) para ayudar a operar un estacionamiento. Tiene un solo botón, en el que el empleado hace clic a medida que un automóvil entra. El programa lleva la cuenta del número de automóviles en el estacionamiento y lo muestra en un cuadro de mensaje.

En esencia, necesitamos sumar 1 a una variable (a la cual llamaremos `cuentaAutos`) en el método `actionPerformed` asociado con el clic del botón. Sin embargo, es importante mencionar que una variable local (declarada dentro del método `actionPerformed`) no funcionará. Las variables locales son temporales; se crean al momento de entrar a un método y se destruyen cuando el método termina. No se preserva el valor que contienen.

He aquí el código correcto (omitimos las partes de la interfaz de usuario en aras de la claridad):

```
public class ContadorAutos extends JFrame
 implements ActionListener {

 private int cuentaAutos = 0;

 public void actionPerformed(ActionEvent event) {
 cuentaAutos = cuentaAutos + 1;
 JOptionPane.showMessageDialog(null, "Autos:" + cuentaAutos);
 }
}
```

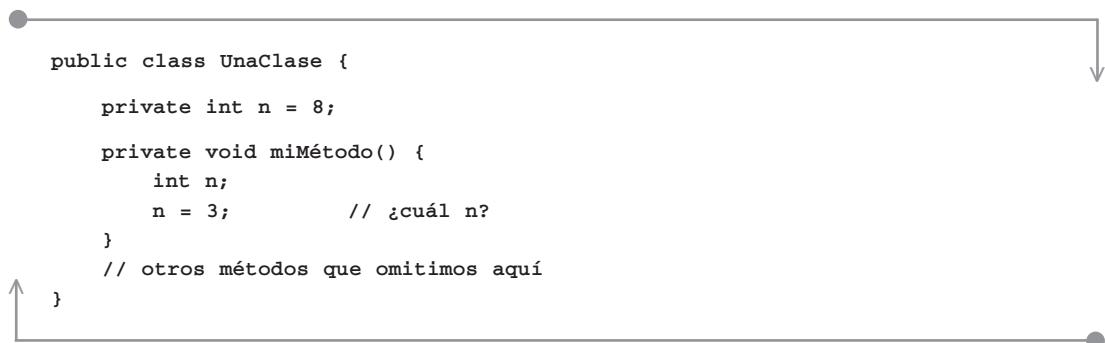


Figura 6.1 Pantalla del programa `ContadorAutos`.

La cuestión aquí es la declaración de la variable `cuentaAutos`:

- Esta variable se declara **fueras** del método, pero **dentro** de la clase `ContadorAutos`. **Cualquier** método de la clase la puede utilizar (aunque aquí sólo la usamos en `actionPerformed`).
- Se ha declarado como **private**, lo cual significa que cualquier otra clase que pudiéramos llegar a tener no podrá utilizarla. La variable está *encapsulada* o sellada dentro de `ContadorAutos`; es decir, es para que la utilicen los métodos de `ContadorAutos` solamente.
- `cuentaAutos` es un ejemplo de una *variable de instancia*. Pertece a una instancia de una clase, en vez de pertenecer a un método. Otro término para ella es *variable a nivel de clase*.
- Se dice que `cuentaAutos` tiene *alcance de clase*. El alcance de un elemento es el área del programa en donde se puede utilizar. El otro tipo de alcance que hemos visto es el alcance local, el cual se utiliza con las variables locales que se declaran dentro de los métodos.
- El estilo preferido para las variables de instancia es declararlas como **private**.
- La convención de Java es poner en minúscula la primera letra de una variable de instancia.

El programador tiene la libertad de elegir los nombres para las variables de instancia. Pero, ¿qué pasa si un nombre coincide con el nombre de una variable local, como en el siguiente ejemplo?



Aunque ambas variables son accesibles (en alcance) dentro de `miMétodo`, la regla es que se elige la variable local. La variable de instancia (a nivel de clase) `n` permanece en **8**.

## PRÁCTICA DE AUTOEVALUACIÓN

- 6.1 En la clase `UnaClase` anterior, ¿cuáles son las consecuencias de eliminar la declaración local de `n`?

Las variables de instancia son esenciales, pero no debemos ignorar a las variables locales. Por ejemplo, si una variable se utiliza sólo dentro de un método y no necesitamos mantener su valor entre una llamada y otra, es mejor hacerla local.

He aquí el programa **ContadorAutos** completo, incluyendo el código de la interfaz de usuario:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class ContadorAutos extends JFrame
 implements ActionListener {

 private int cuentaAutos = 0;

 private JButton botón;

 public static void main(String[] args) {
 ContadorAutos marco = new ContadorAutos();
 marco.setSize(300, 200);
 marco.createGUI();
 marco.setVisible(true);
 }

 private void createGUI() {
 setDefaultCloseOperation(EXIT_ON_CLOSE);
 Container ventana = getContentPane();
 ventana.setLayout(new FlowLayout());

 botón = new JButton("Entra un auto");
 ventana.add(botón);
 botón.addActionListener(this);
 }

 public void actionPerformed(ActionEvent event) {
 cuentaAutos = cuentaAutos + 1;
 JOptionPane.showMessageDialog(null, "Autos:" + cuentaAutos);
 }
}
```

No caiga en la tentación de enmendar el código de la interfaz de usuario por ahora. Debe estar exactamente como se muestra.

Ahora que introducimos el alcance **private**, vamos a aplicarlo al código de la interfaz de usuario. Los elementos de una ventana (como los botones) necesitan estar ahí durante la vida del programa. Además, es común que varios métodos los utilicen. Por estas razones se declaran como variables de instancia, fuera de cualquier método. Podemos ver esto en acción en la clase **ContadorAutos**, en donde se declara un botón de la siguiente forma:

```
private JButton botón;
```

en la misma área de código que la variable **cuentaAutos**. Más adelante en este capítulo retomaremos la explicación sobre las clases de la interfaz de usuario.

## PRÁCTICA DE AUTOEVALUACIÓN

- 6.2 ¿Qué hace el siguiente programa? (Omitimos de manera intencional la creación de los objetos de la GUI para que se pueda enfocar en los alcances).

```
private int x = 0;

public void actionPerformed(ActionEvent event) {
 Graphics paper = panel.getGraphics();
 Graphics papel = panel.getGraphics();
 x = x + 10;
}
```

### ● Instanciación: uso de constructores con new

Hasta aquí, hemos escrito programas que utilizan los tipos `int` y `double`. A éstos se les considera como tipos “integrados” o “primitivos”: `no` son instancias de clases (es decir, no son objetos). Recuerde que podemos declararlos y proveer un valor inicial, como en:

```
int n = 3;
```

Lo que estamos diciendo es: “crea un nuevo entero llamado `n`, con un valor inicial de 3”.

Sin embargo, hemos usado otros tipos de elementos (como botones y áreas para dibujar gráficos). Éstos `son` instancias de clases. Tenemos que crearlos de una manera especial, mediante la palabra `new`. Al proceso de crear una instancia con `new` se le conoce como *instanciación*.

Para ilustrar el uso de `new` vamos a estudiar la clase `Random` de la biblioteca de Java.

### ● La clase Random

Los números aleatorios son muy útiles en simulaciones y juegos; por ejemplo, podemos proporcionar al jugador una situación inicial distinta cada vez que juegue. Las instancias de la clase `Random` nos proporcionan un “flujo” de números, los cuales podemos obtener uno a la vez mediante el método `nextInt`. A continuación veremos un programa ([LíneasAleatorias](#)) que dibuja una línea aleatoria cada vez que hacemos clic en el botón. Un extremo de la línea está fijo en (0, 0) y el otro extremo tiene una posición `x` e `y` aleatoria. Antes de dibujar la línea borramos el área de dibujo, para lo cual dibujamos un rectángulo blanco que abarque toda el área de dibujo (en este ejemplo, de 100 por 100) y después establecemos el color en negro. En la figura 6.2 se muestran dos pantallas y he aquí el código:

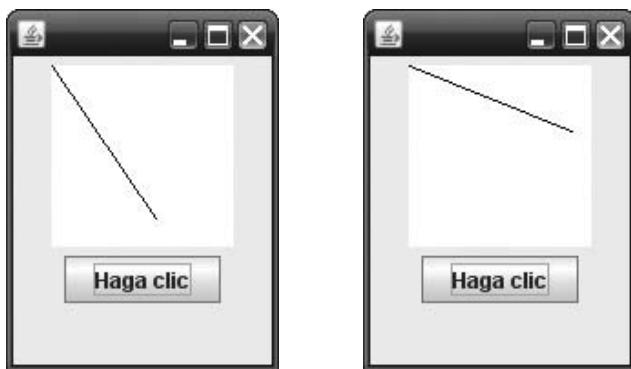


Figura 6.2 Dos pantallas del programa LíneasAleatorias.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.util.*;

public class LíneasAleatorias extends JFrame
 implements ActionListener {

 private Random posicionesAleatorias = new Random();
 private JButton botón;
 private JPanel panel;

 public static void main(String[] args) {
 LíneasAleatorias marco = new LíneasAleatorias();
 marco.setSize(150, 200);
 marco.crearGUI();
 marco.setVisible(true);
 }

 private void crearGUI() {
 setDefaultCloseOperation(EXIT_ON_CLOSE);
 Container ventana = getContentPane();
 ventana.setLayout(new FlowLayout());

 panel = new JPanel();
 panel.setPreferredSize(new Dimension(100,100));
 panel.setBackground(Color.white);
 ventana.add(panel);

 botón = new JButton("Haga clic");
 ventana.add(botón);
 botón.addActionListener(this);
 }
}
```

```
public void actionPerformed(ActionEvent event) {
 int xFinal, yFinal;
 Graphics papel = panel.getGraphics();

 papel.setColor(Color.white);
 papel.fillRect(0, 0, 100, 100);
 papel.setColor(Color.black);

 xFinal = posicionesAleatorias.nextInt(100);
 yFinal = posicionesAleatorias.nextInt(100);
 papel.drawLine(0, 0, xFinal, yFinal);
}
}
```

Para usar `Random` de una manera conveniente, necesitamos esta instrucción de importación:

```
import java.util.*;
```

Si omitiéramos la instrucción de importación tendríamos que hacer referencia a la clase de esta manera:

```
java.util.Random
```

El uso de `import` nos permite ser más directos.

Después debemos declarar e inicializar una instancia de nuestra clase. Esto podemos hacerlo de dos maneras. Uno de los métodos sería utilizar una sola instrucción, como en el siguiente ejemplo:

```
private Random posicionesAleatorias = new Random();
```

Observe que:

- Elegimos el alcance `private` en vez del local.
- Después de `private` indicamos la clase del elemento que vamos a declarar. En este caso el elemento es una instancia de la clase `Random`.
- Elegimos el nombre `posicionesAleatorias` para nuestra instancia. Otros nombres apropiados para nuestra instancia serían `númerosAleatorios`, `aleatorio`. También existe la posibilidad de usar `random` como el nombre de una instancia. Java es susceptible al uso de mayúsculas y minúsculas; la convención es que todos los nombres de las clases empiecen con mayúscula. Por lo tanto, `random` (con minúscula) se puede usar para el nombre de una instancia. En un programa en donde sólo vamos a crear una instancia a partir de una clase, es común usar el mismo nombre que la clase (pero con la primera letra minúscula). En este ejemplo, `posicionesAleatorias` transmite el significado deseado. Utilizamos plural debido a que nuestra instancia puede proveer todos los números aleatorios que necesitemos.
- La palabra `new` se antepone al uso del constructor, que en esencia es un método con el mismo nombre que la clase: `Random`. El uso de `new` crea una nueva instancia de una clase en la memoria de acceso aleatorio, y la asigna a `posicionesAleatorias`.
- Los constructores se pueden sobrecargar, por lo que necesitamos elegir el constructor más conveniente. `Random` tiene dos constructores con distintos parámetros; en este caso es apropiado usar el que no tiene parámetros.
- Podemos considerar que la instrucción consta de dos partes:

```
private Random posicionesAleatorias...
```

y:

```
... = new Random();
```

La primera parte declara a **posicionesAleatorias** como una variable de la clase **Random**, pero aún no tiene asociada una instancia concreta. La segunda parte llama al constructor de la clase **Random** para completar las tareas de declaración e inicialización.

Otra forma de declarar e inicializar instancias es mediante instrucciones de declaración e inicialización en distintas áreas del programa, como en el siguiente ejemplo:

```
public class LíneasAleatorias extends JFrame
 implements ActionListener {

 private Random posicionesAleatorias;

 ...

 private void unMetodo() {
 posicionesAleatorias = new Random();
 ...
 }

}
```

Sin importar el método que seleccionemos, debemos tener en cuenta varios puntos:

- La declaración establece la clase de la instancia. En este caso es una instancia de **Random**.
- La declaración establece el alcance de la instancia. En este caso, **posicionesAleatorias** tiene alcance de clase; se puede utilizar en cualquier método de la clase **LíneasAleatorias** en vez de ser local para un método.
- **posicionesAleatorias** es privada. No se puede utilizar en otras clases que no sean nuestra clase **LíneasAleatorias**. Por lo general, todas las variables de instancia las hacemos privadas.
- La forma de declaración de una sola instrucción es conveniente, pero no siempre se utiliza. Algunas veces necesitamos inicializar una variable de instancia en una etapa posterior, con base en los valores calculados por el programa. Esto se debe hacer dentro de un método, aun cuando la declaración se coloque fuera de los métodos. Más adelante en el capítulo mostraremos ejemplos sobre cómo separar la inicialización de la declaración.

## PRÁCTICA DE AUTOEVALUACIÓN

6.3 ¿Cuál es el error en el siguiente fragmento de código?

```
public class SomeClass extends JFrame
 implements ActionListener {

 private Random r;
 r = new Random();
 ...
}
```

Ahora regresemos al programa `LíneasAleatorias`. Hasta aquí sólo hemos creado un objeto; es decir, una instancia de la clase `Random` llamada `posicionesAleatorias`. Aún no hemos creado números aleatorios reales.

Una vez que creamos un objeto con `new`, podemos utilizar sus métodos. La documentación nos indica que hay varios métodos que nos proporcionan un número aleatorio, por lo que elegimos el método que provee enteros y nos permite especificar el rango de los números. Este método se llama `nextInt` (debido a que obtiene el siguiente número aleatorio de una secuencia de números). En nuestro programa colocamos lo siguiente:

```
xFinal = posicionesAleatorias.nextInt(100);
yFinal = posicionesAleatorias.nextInt(100);
```

Se eligió el rango de números aleatorios (100 en este caso) de manera que fuera adecuado para el tamaño del área de dibujo.

Para finalizar, declaramos una instancia de la clase apropiada (`Random`) y utilizamos `new` para crearla e inicializarla. Se pueden combinar o separar estas dos etapas, dependiendo del programa específico en el que estemos trabajando. Después utilizamos el método `nextInt` de la instancia creada.

Vamos a ampliar nuestro estudio sobre la clase `Random`. Analizaremos sus constructores y sus métodos más útiles.

En primer lugar hay dos constructores. En el ejemplo anterior usamos el que no tiene parámetros. Sin embargo, el constructor está sobrecargado: hay otra versión con un solo parámetro. He aquí los dos constructores en uso:

```
private Random random = new Random();
private Random randomIgual = new Random(1000);
```

La primera versión produce una secuencia aleatoria distinta cada vez que ejecutamos el programa. La segunda versión deriva la secuencia aleatoria del número que suministramos, que puede ser cualquier valor. En este caso ocurre la misma secuencia aleatoria cada vez. Podríamos usar esta segunda forma si quisieramos realizar muchas ejecuciones de prueba con la misma secuencia.

La figura 6.3 muestra los métodos más útiles.

Vamos a considerar el método `nextInt` con más detalle. En el programa `LíneasAleatorias` utilizamos las siguientes líneas:

```
xFinal = posicionesAleatorias.nextInt(100);
yFinal = posicionesAleatorias.nextInt(100);
```

|                             |                                                                                 |
|-----------------------------|---------------------------------------------------------------------------------|
| <code>nextInt(int n)</code> | Devuelve un valor <code>int &gt;= 0 y &lt; n</code>                             |
| <code>nextDouble()</code>   | No tiene parámetros. Devuelve un valor <code>double &gt;= 0.0 y &lt; 1.0</code> |

**Figura 6.3** Métodos de la clase `Random`.

Esto producirá un valor aleatorio en el rango de `0` a `99`. El valor `100` nunca ocurrirá. Esto se debe a que la especificación de `nextInt` indica “menor que” en vez de “menor o igual que”. Ésta es una causa común de los errores de programación, debido a que la mayoría de los problemas se declaran con rangos inclusivos, como en “al lanzar un dado se obtiene un número del `1` al `6`”, o “las cartas del juego se enumeran del `2` al `10`, excluyendo los ases”. Se aplican advertencias similares para `nextDouble`, que nunca producirá un valor exacto de `1.0`.

Ahora vamos a escribir un método que simplifica el uso de los números aleatorios. Tiene dos parámetros que nos permiten especificar los valores mínimos y máximos, ambos incluidos, de nuestros números. He aquí el código:

```
private int aleatorioEnRango(int min, int max) {
 return min+random.nextInt(max-min+1);
}
```

Para simular el lanzamiento de un dado, podríamos usar lo siguiente:

```
int suertudo;
suertudo = aleatorioEnRango(1, 6);
```

Al usar una clase, es importante comprender las herramientas que proporcionan sus métodos y constructores. Algunas veces la documentación que se incluye en los sistemas de Java es bastante difícil de comprender, por lo que en el apéndice A sintetizamos todas las clases que utilizamos en este libro.

## PRÁCTICA DE AUTOEVALUACIÓN

**6.4** ¿Cómo podríamos llamar a `aleatorioEnRango` para obtener una edad aleatoria en el rango de `16` a `59`, ambos incluidos?

### ● El método `main` y `new`

Ya hemos visto el uso de `new` para crear una nueva instancia de una clase, la cual posteriormente podemos usar a través de sus métodos. Pero si hacemos memoria y regresamos a los detalles de sus programas, podrá ver que todos ellos son clases de la forma:

```
public class UnNombre...{
 private declaraciones...
 una serie de métodos...
}
```

En sentido informal hemos hablado de “escribir un programa”, aunque de hecho deberíamos decir “escribir una clase”. Pero, ¿es el programa una clase, o una instancia de una clase?

Recuerde que el sistema de Java llama de manera automática al método `main` antes de que ocurra cualquier otra cosa. Analice cualquiera de nuestros métodos `main`. Su primera tarea es usar `new` para crear una instancia de la clase que lo contiene.

Vamos a seguir hablando de programas, ya que esto es más natural. Para responder nuestra pregunta sobre los programas y las clases: un programa en ejecución es una instancia de una clase. Se genera con la instrucción `new` dentro del método `main`.

## El kit de herramientas Swing

Al crear Java se incluyó un conjunto de clases que contienen componentes de interfaz de usuario, como botones, barras de desplazamiento, etc. A este conjunto de clases se le denominó Kit de herramientas de Ventanas Abstractas (AWT). Sin embargo, algunos de los componentes eran de una calidad bastante inferior; además, se veían diferentes en distintas plataformas debido a que utilizaban los componentes que proveía el sistema operativo en uso. En el apéndice B se incluyen las generalidades sobre el AWT.

Con el propósito de mejorar esta situación, se escribió un conjunto de componentes en Java para proveer más herramientas que se vieran idénticas en cualquier plataforma. A estas clases se les denominó kit de herramientas Swing. En los ejercicios del libro hemos usado con frecuencia la clase `JButton`; la `J` indica que la clase está escrita en Java.

Aunque Swing ofrece más poder, de todas formas necesitamos partes del antiguo AWT, como puede ver en las instrucciones `import` que colocamos en la parte superior de nuestros programas.

## Eventos

En secciones anteriores vimos la clase `Random`, cómo crear nuevas instancias y manipularlas con sus métodos. Seguiremos esta metodología para muchas clases. Sin embargo, hay otras clases (como `JButton`) que son distintas, ya que involucran eventos. Vamos a analizar esta clase con detalle y después generalizaremos el uso de las clases, para que usted pueda usar cualquier otra clase con la que se encuentre.

Hemos usado eventos en muchos de los programas que hemos visto. Creamos un botón y colocamos el código en el método `actionPerformed` para responder al evento. Aquí hablaremos sobre los eventos con más detalle.

En Java, los eventos se dividen en categorías. Por ejemplo, tenemos:

- Eventos de “acción”, como hacer clic en un botón.
- Eventos de “cambio”, como ajustar la posición de un control deslizable para cambiar el volumen del altavoz de una computadora.

Recordemos la siguiente línea del programa `ContadorAutos`:

```
public class ContadorAutos extends JFrame
 implements ActionListener {
```

La palabra clave `extends` expresa herencia, tema que veremos en el capítulo 10. La palabra clave `implements` se puede usar para proveer el manejo de eventos. He aquí una analogía. Suponga que tiene una tarjeta SuperCrédito y que ve un anuncio afuera de una tienda, el cual menciona que se aceptan tarjetas SuperCrédito. Usted supone que al hacer una compra la tienda le proporcionará las herramientas que necesita, como una máquina adecuada para procesar su tarjeta de crédito. En otras palabras, la tienda implementa la interfaz de SuperCrédito. En el capítulo 23 hablaremos sobre las interfaces.

Al usar `implements ActionListener`, estamos declarando que nuestro programa implementa la interfaz `ActionListener`. Para ello debemos proveer un método llamado `actionPerformed`, el cual se llamará cuando ocurra un evento de acción, como el clic de un botón.

Además, tenemos que registrar el programa como un “escuchador” (o “detector”) para los tipos de eventos. A continuación veremos cómo hacer esto.

## ● Creación de un objeto JButton

Aquí vamos a ver el proceso de crear un botón. Al igual que cuando usamos la clase `Random`, debemos declarar e inicializar el objeto antes de usarlo. Además, debemos implementar la interfaz `ActionListener` y registrar el programa como escuchador para los eventos de acción. He aquí otra vez el programa `ContadorAutos` para analizarlo:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class ContadorAutos extends JFrame
 implements ActionListener {
 private int cuentaAutos = 0;
 private JButton botón;

 public static void main(String[] args) {
 ContadorAutos marco = new ContadorAutos();
 marco.setSize(300, 200);
 marco.crearGUI();
 marco.setVisible(true);
 }

 private void crearGUI() {
 setDefaultCloseOperation(EXIT_ON_CLOSE);
 Container ventana = getContentPane();
 ventana.setLayout(new FlowLayout());

 botón = new JButton("Entra un auto");
 ventana.add(botón);
 botón.addActionListener(this);
 }

 public void actionPerformed(ActionEvent event) {
 cuentaAutos = cuentaAutos + 1;
 JOptionPane.showMessageDialog(null, "Autos:" + cuentaAutos);
 }
}
```

El proceso es el siguiente:

- Primero declaramos que nuestro programa implementa la interfaz `ActionListener`:

```
public class ContadorAutos extends JFrame
 implements ActionListener {
```

Para ello tenemos que escribir un método llamado `actionPerformed`, como se muestra en el código.

- A continuación declaramos el botón, como una variable de instancia:

```
private JButton botón;
```

- Despues creamos el botón y proveemos el texto que aparecerá dentro de él en su constructor. Podemos hacer esto al mismo tiempo que la declaración, pero en este caso elegimos agrupar todo el código relacionado con la inicialización en un método, al cual llamamos `crearGUI`. La inicialización es:

```
botón = new JButton("Entra un auto");
```

- El siguiente paso es agregar un botón a una instancia de la clase `Container` llamada `ventana`:

```
ventana.add(botón);
```

Cabe mencionar que `add` es un método de `ventana` y no de `botón`. El método `add` coloca los elementos en la pantalla, en orden de izquierda a derecha. Los elementos en una fila están centrados. Cuando se llena una fila, empieza de manera automática una nueva fila de objetos (a este esquema se le conoce como “esquema de flujo”. Hay otro esquema conocido como “esquema de borde”, que veremos en el apéndice A).

- A continuación registramos el programa como escuchador para los eventos de acción provenientes del botón:

```
botón.addActionListener(this);
```

- He aquí el método manejador de eventos, en donde colocamos el código para responder al evento:

```
public void actionPerformed(ActionEvent event) {
```

Como puede ver, el proceso es bastante complicado. La buena noticia es que el proceso es casi idéntico cada vez que lo utilizamos.

Hay ciertos elementos arbitrarios que elegimos en el programa. Por ejemplo:

- El nombre de la instancia de `JButton`; elegimos `botón`.
- El texto que se muestra en el botón; elegimos `Entra un auto`.

Hay varias partes esenciales del programa que no podemos cambiar:

- Las instrucciones `import`.
- La instrucción `implements ActionListener`.
- Un método `actionPerformed`.
- El uso de `addActionListener` para registrarse como escuchador del botón.
- El método `main` y las partes de `crearGUI` que establecen el cierre del marco exterior y la disposición de los objetos.

Al decir que no podemos cambiar estas partes, hablamos en serio. No invente sus propios nombres, como `clickPerformed`.

## ● Lineamientos para usar objetos

Ya vimos cómo se pueden incorporar a los programas las instancias de la clase `Random` y la clase `JButton`. Ahora estamos en posición de retroceder y proveerle algunos lineamientos generales. Despues aplicaremos estos lineamientos a las clases `JLabel`, `JTextField`, `JSlider`, `JPanel`, `Timer` e `ImageIcon`.

He aquí la metodología a seguir:

1. Examine la documentación de la clase en cuestión. Determine la instrucción `import` requerida. Algunas veces habrá que importar clases adicionales, así como la clase en cuestión (como en el caso de `JSlider` que veremos más adelante).
2. Seleccione un constructor para usarlo con `new`.
3. Si la clase es un componente de una interfaz de usuario, agréguela a la ventana.
4. Una vez que haya declarado y creado la instancia con `new`, úsela a través de sus métodos.

Ahora examinaremos otras clases útiles. No hay nuevas herramientas de Java requeridas; hemos visto cómo incorporar clases y crear instancias. Sin embargo, estas clases son muy útiles y aparecen en muchos de los siguientes capítulos.

## ● La clase `JLabel`

La clase `JLabel` nos permite mostrar texto que no cambia, como instrucciones o indicadores para el usuario. Si deseamos texto que cambie (como el resultado de un cálculo) debemos usar un objeto `JTextField`. El programa `SumarCamposTexto` (figura 6.4) muestra cómo se usa para visualizar el carácter = fijo. Ahora examinemos su uso:

- Es un componente de Swing, por lo que basta con usar nuestra instrucción `import` normal.
- He aquí un ejemplo de su constructor, en el cual proveemos el texto a mostrar:

```
etiquetaIgual = new JLabel(" = ");
```



Figura 6.4 Pantalla del programa `SumarCamposTexto`.

- Se agrega a la ventana en forma similar a un botón.
- No produce eventos y es poco probable que el programa lo manipule de nuevo, una vez que se agregue a la ventana.

He aquí el código de **SumarCamposTexto**, que suma dos valores al hacer clic en el botón +:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class SumarCamposTexto extends JFrame
 implements ActionListener {

 private JTextField campoNúmero1, campoNúmero2, campoSuma;
 private JLabel etiquetaIgual;
 private JButton botónSuma;

 public static void main(String[] args) {
 SumarCamposTexto marco = new SumarCamposTexto();
 marco.setSize(350, 100);
 marco.crearGUI();
 marco.setVisible(true);
 }

 private void crearGUI() {
 setDefaultCloseOperation(EXIT_ON_CLOSE);
 Container ventana = getContentPane();
 ventana.setLayout(new FlowLayout());

 campoNúmero1 = new JTextField(7);
 ventana.add(campoNúmero1);

 botónSuma = new JButton("+");
 ventana.add(botónSuma);
 botónSuma.addActionListener(this);

 campoNúmero2 = new JTextField(7);
 ventana.add(campoNúmero2);

 etiquetaIgual = new JLabel(" = ");
 ventana.add(etiquetaIgual);

 campoSuma = new JTextField(7);
 ventana.add(campoSuma);
 }

 public void actionPerformed(ActionEvent event) {
 int número1 = Integer.parseInt(campoNúmero1.getText());
 int número2 = Integer.parseInt(campoNúmero2.getText());
 campoSuma.setText(Integer.toString(número1 + número2));
 }
}
```

## ● La clase JTextField

La clase **JTextField** provee un área de una sola línea que se puede usar para mostrar o introducir texto. Examinemos su uso:

- Es un componente de Swing, por lo que basta con usar nuestra instrucción **import** normal.
- Cuando el usuario oprime la tecla “Intro” en un campo de texto, se produce un evento de acción. Si deseamos usar este evento, debemos implementar **ActionListener** y proveer un método **actionPerformed**. En nuestro ejemplo utilizaremos un clic de botón en vez de la tecla “Intro” para iniciar el cálculo.
- He aquí ejemplos de sus constructores:

```
campoTexto1 = new JTextField(15);
campoTexto2 = new JTextField("¡Hola!", 15);
```

El primero crea un campo de texto vacío con la anchura especificada (en términos de caracteres) y el segundo también nos permite configurar cierto texto inicial. Cabe mencionar que la fuente predeterminada de los campos de texto es proporcional, por lo que **m** ocupa más espacio que **1**. La anchura de un campo de texto se basa en el tamaño de una **m**.

- Se agrega a la ventana en forma similar a un botón.
- Su contenido se puede manipular mediante los métodos **setText** y **getText**, como en el siguiente ejemplo:

```
String s;
s = campoTexto1.getText();
campoTexto1.setText(s);
```

En el programa **SumarCamposTexto**, el usuario introduce dos enteros en los campos de texto de la izquierda. Al hacer clic en el botón, aparece la suma en el tercer campo de texto. Recuerde la forma en que usamos los cuadros de entrada para introducir números en el capítulo 4. De nuevo, necesitamos convertir en entero la cadena introducida en el campo de texto y, a su vez, convertir en cadena nuestro resultado entero para poder mostrarlo en pantalla. Usaremos los métodos **Integer.parseInt** e **Integer.toString**. He aquí el código:

```
int número1 = Integer.parseInt(campoNúmero1.getText());
int número2 = Integer.parseInt(campoNúmero2.getText());
campoSuma.setText(Integer.toString(número1 + número2));
```

## PRÁCTICA DE AUTOEVALUACIÓN

- 6.5** Vuelva a escribir el programa **ContadorAutos** de manera que se muestre el conteo en un campo de texto en vez de un cuadro de mensaje.

## ● La clase JPanel

El panel se puede usar para dibujar o para contener otros objetos, como los botones. Al crear un área de dibujo, por lo general necesitamos especificar su tamaño en píxeles en vez de dejar que Java la trate de la misma forma que a los botones.

He aquí el código estándar que usamos para crear un panel con un tamaño específico:

```
panel = new JPanel();
panel.setPreferredSize(new Dimension(200, 200));
panel.setBackground(Color.white);
ventana.add(panel);
```

Para usar el panel como área de dibujo en vez de como contenedor para otros objetos, escribimos el siguiente código:

```
Graphics papel = panel.getGraphics();
```

## ● La clase Timer

El temporizador crea eventos de acción espaciados en forma regular, lo cual podemos considerar como el tic de un reloj. Podemos iniciar y detener el temporizador, además de controlar su velocidad. A diferencia de un botón, el temporizador no tiene representación en pantalla. He aquí las principales herramientas del temporizador:

- El temporizador crea tics a intervalos regulares. Cada tic es un evento manejado por el método `actionPerformed`.
- Hay que realizar con cuidado la importación. Hay dos clases `Timer` en estas bibliotecas:

```
java.util
javax.swing
```

Nosotros queremos la que está en la biblioteca de Swing. Si importamos todas las clases de cada biblioteca y después tratamos de declarar una instancia de la clase `Timer` se producirá un error de compilación. He aquí cómo podemos resolver el conflicto:

- La mayoría de nuestros programas importan todas las clases de `javax.swing`. Si el programa no necesita la biblioteca `java.util`, no hay problema. Declaramos un temporizador de la siguiente manera:

```
private Timer temporizador;
```

- Si el programa necesita ambas bibliotecas (como en el programa `GotasDeLluvia` que veremos a continuación), entonces podemos declarar un temporizador así:

```
private javax.swing.Timer temporizador;
```

Debemos recurrir a esta forma larga cada vez que usemos el nombre de la clase `Timer`.

- El temporizador crea eventos de acción, por lo que especificamos `implements ActionListener` y proveemos un método `actionPerformed`.

- El constructor para el temporizador requiere dos parámetros:
  - Un entero que especifique el número de milisegundos entre eventos (tics).
  - El programa que se registra para detectar los eventos de acción. Al igual que con los botones, usamos `this`. He aquí un ejemplo:

```
temporizador = new Timer(1000, this);
```

- Podemos iniciar y detener el temporizador mediante los métodos `start` y `stop`.
- El tiempo entre eventos (en milisegundos) se puede modificar mediante `setDelay`, como en:

```
temporizador.setDelay(500);
```

He aquí un programa (`EjemploTimer`) que muestra los minutos y segundos en la pantalla. La figura 6.5 muestra la pantalla resultante y he aquí el código.



Figura 6.5 Pantalla del programa `EjemploTimer`.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class EjemploTimer extends JFrame
 implements ActionListener {

 private JTextField campoSegs, campoMins;
 private JLabel etiquetaSegs, etiquetaMins;
 private int tics = 0;
 private Timer temporizador;

 public static void main (String[] args) {
 EjemploTimer marco = new EjemploTimer();
 marco.setSize(300,100);
 marco.crearGUI();
 marco.setVisible(true);
 }

 private void crearGUI() {
 setDefaultCloseOperation(EXIT_ON_CLOSE);
 Container ventana = getContentPane();
 ventana.setLayout(new FlowLayout());
```

```

etiquetaMins = new JLabel("Mins: ");
ventana.add(etiquetaMins);
campoMins = new JTextField(2);
ventana.add(campoMins);
etiquetaSegs = new JLabel(" Segs: ");
ventana.add(etiquetaSegs);
campoSegs = new JTextField(2);
ventana.add(campoSegs);
temporizador = new Timer(1000, this);
temporizador.start();
}
}

public void actionPerformed(ActionEvent event) {
campoMins.setText(Integer.toString(tics / 60));
campoSegs.setText(Integer.toString(tics % 60));
ticks = ticks + 1;
}
}

```

Creamos un temporizador con un evento cada segundo (1000 milisegundos). El manejo del evento implica lo siguiente:

- Una variable privada (`tics`) para contar el número de tics.
- Calcular los minutos mediante una división entre 60.
- Usar % para evitar que el despliegue de los segundos pase de 59.
- Actualizar los campos de texto.
- Incrementar el contador.

## PRÁCTICA DE AUTOEVALUACIÓN

**6.6** Explique por qué la variable `tics` no puede ser local.

### ● La clase `JSlider`

Nuestra última clase es la del control deslizable. Esta clase provee un control que se puede arrastrar para seleccionar un valor. Podemos verlo en acción en el programa `GotasDeLluvia` de la figura 6.6. He aquí los puntos principales:

- Necesitamos importar la biblioteca `javax.swing.event` para el manejo de eventos.
- Crea eventos de “cambio”. Necesitamos usar `implements ChangeListener` y proveer un método `stateChanged` para manejar los eventos.
- Para crear un control deslizable se requiere proveer cuatro parámetros al constructor:
  - La orientación del control deslizable, que se especifica mediante `JSlider.VERTICAL` o `JSlider.HORIZONTAL`.



Figura 6.6 Pantalla del programa `GotasDeLluvia`.

- El valor mínimo del control deslizable.
- El valor máximo del control deslizable.
- La posición inicial del deslizador.
- El valor actual se obtiene mediante el método `getValue`.
- El método `stateChanged` se llama cuando el usuario mueve el deslizador. Se crearán varios eventos al momento de arrastrar el deslizador, pero la llamada final ocurrirá cuando el usuario elija un valor.

He aquí el código de `GotasDeLluvia`. En la figura 6.6 aparece la pantalla resultante:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
import java.util.*;

public class GotasDeLluvia extends JFrame implements
 ActionListener, ChangeListener {

 private JPanel panel;
 private Random aleatorio;
 private javax.swing.Timer temporizador;
 private JSlider deslizable;
 private JTextField campointervalo;
 private JLabel etiquetaIntervalo;
```

```
public static void main (String[] args) {
 GotasDeLluvia marco = new GotasDeLluvia();
 marco.setSize(250, 300);
 marco.createGUI();
 marco.setVisible(true);
}

private void createGUI() {
 setDefaultCloseOperation(EXIT_ON_CLOSE);
 Container ventana = getContentPane();
 ventana.setLayout(new FlowLayout());
 aleatorio = new Random();

 panel = new JPanel();
 panel.setPreferredSize(new Dimension(200, 200));
 panel.setBackground(Color.white);
 ventana.add(panel);

 etiquetaIntervalo = new JLabel("Intervalo de tiempo: ");
 ventana.add(etiquetaIntervalo);
 campoIntervalo = new JTextField(10);
 ventana.add(campoIntervalo);

 deslizable = new JSlider(JSlider.HORIZONTAL, 200, 2000, 1000);
 ventana.add(deslizable);
 deslizable.addChangeListener(this);
 campoIntervalo.setText(Integer.toString(deslizable.getValue()));
 temporizador = new javax.swing.Timer(1000, this);
 temporizador.start();
}

public void actionPerformed(ActionEvent event) {
 int x, y, tamaño;
 Graphics papel = panel.getGraphics();
 x=aleatorio.nextInt(200);
 y = aleatorio.nextInt(200);
 tamaño = aleatorio.nextInt(20);
 papel.fillOval(x,y, tamaño, tamaño);
}

public void stateChanged(ChangeEvent e) {
 int timeGap = deslizable.getValue();
 campoIntervalo.setText(Integer.toString(timeGap));
 temporizador.setDelay(timeGap);
}

}
```

El programa simula gotas de lluvia de un tamaño aleatorio que caen en una hoja de papel. El usuario puede modificar el tiempo entre cada gota con sólo mover el deslizador.

Cada vez que ocurre un evento del temporizador, el programa dibuja un círculo de tamaño aleatorio en una posición aleatoria. Al mover el deslizador aparece el valor actual del control en un campo de texto y se modifica la velocidad del temporizador. Descubrimos el rango del control des-

lizable de 200 a 2000 mediante la experimentación. El programa usa la mayoría de las clases que hemos examinado, pero hay dos nuevos puntos:

- Declaramos el temporizador con la forma completa de su nombre, ya que se importó la biblioteca `util` para `Random`.
- Explotamos la interdependencia de los componentes durante la fase de inicialización. Establecemos el valor inicial del campo de texto con el valor inicial del control deslizable.

## ● La clase `ImageIcon` – cómo mover una imagen

La clase `ImageIcon` nos permite mostrar una imagen en la pantalla, como se muestra en la figura 6.7. La imagen puede provenir de una variedad de fuentes, como una cámara digital, un escáner o un paquete de dibujo. Sin importar cuál sea su origen, la imagen se debe almacenar en un archivo antes de mostrarla con Java. La clase `ImageIcon` ofrece soporte para una variedad de tipos de archivos, incluyendo `jpeg` (o `jpg`) y `gif`. Vamos a examinar su uso:

- Es un componente de Swing, por lo que no se requiere ninguna instrucción de importación adicional.
- Su constructor tiene un solo parámetro de cadena, que representa el nombre del archivo que contiene la imagen. He aquí un ejemplo:

```
ImageIcon miImagen = new ImageIcon("miFoto.jpg");
```

En el ejemplo anterior suponemos que la imagen está guardada en la misma carpeta que el archivo de clase que se va a ejecutar. Si usted utiliza un IDE, explore las carpetas de un proyecto para averiguar en dónde almacena los archivos `.class`.

- Una vez creada la instancia de `ImageIcon` con `new`, podemos usar sus métodos. En nuestro ejemplo usamos el método `paintIcon` para mostrar la imagen, como en:

```
miImagen.paintIcon(this, papel, 10, 10);
```

Los parámetros de `paintIcon` son:

- El objeto actual en ejecución, representado por `this`.
  - El área `Graphics` en donde se va a colocar la imagen.
  - La posición horizontal (x) de la esquina superior izquierda de la imagen dentro del área de dibujo.
  - La posición vertical (y) de la esquina superior izquierda de la imagen dentro del área de dibujo.
- Si la imagen es demasiado grande como para caber dentro del área de dibujo, se recorta. Sólo se muestra la parte que se pueda ajustar dentro del área de dibujo.

La figura 6.7 muestra la pantalla de un programa que despliega una pequeña imagen y después la recorre una pequeña distancia en sentido diagonal, cada vez que el usuario hace clic en el botón.

He aquí el código:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
```

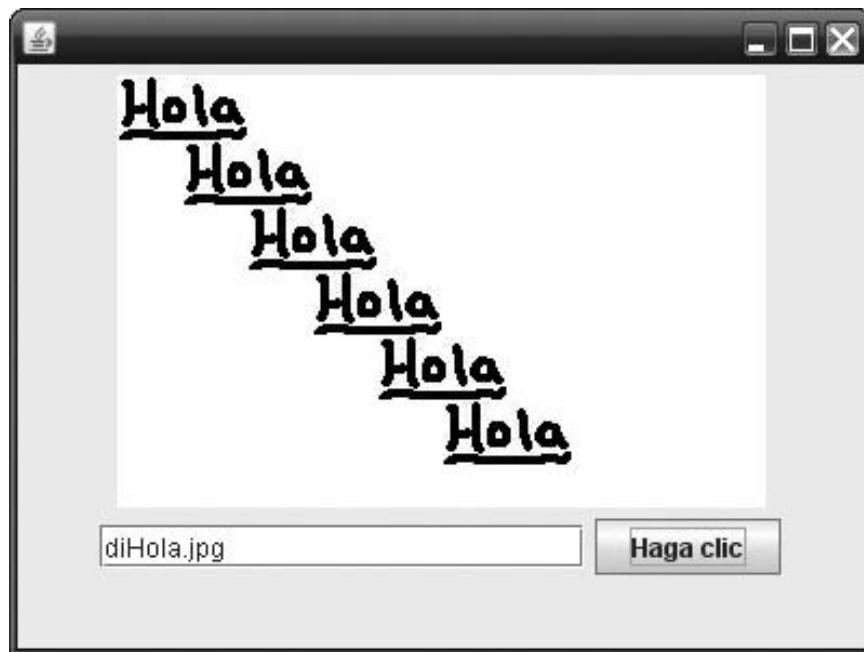


Figura 6.7 Pantalla del programa **ImageDemo**.

```
public class ImageDemo extends JFrame
 implements ActionListener {

 private JButton botón;
 private JPanel panel;
 private JTextField campoArchivo;
 private int posiciónX = 0, posiciónY = 0;

 public static void main (String[] args) {
 DemoImagen marco = new DemoImagen();
 marco.setSize(400, 300);
 marco.crearGUI();
 marco.setVisible(true);
 }

 private void crearGUI() {
 setDefaultCloseOperation(EXIT_ON_CLOSE);
 Container ventana = getContentPane();
 ventana.setLayout(new FlowLayout());

 panel = new JPanel();
 panel.setPreferredSize(new Dimension(300, 200));
 panel.setBackground(Color.white);
 ventana.add(panel);
```

```

campoArchivo = new JTextField(20);
ventana.add(campoArchivo);

botón = new JButton("Haga clic");
ventana.add(botón);
botón.addActionListener(this);
}

public void actionPerformed(ActionEvent event) {
 ImageIcon miImagen = new ImageIcon(campoArchivo.getText());
 Graphics papel = panel.getGraphics();
 miImagen.paintIcon(this, papel, posiciónX, posiciónY);
 posiciónX = posiciónX + 30;
 posiciónY = posiciónY + 30;
}
}

```



El programa requiere que el usuario escriba el nombre de un archivo de imagen en un campo de texto. Es necesario colocar la imagen en la misma carpeta que el archivo de clase del programa (puede crear una pequeña imagen con un programa de dibujo, o puede descargar una).

Al hacer clic en el botón, la imagen se vuelve a dibujar en una posición hacia abajo y a la derecha de su posición actual, para lo cual se suma 30 a las variables **posiciónX** y **posiciónY**.

Considere el alcance de estas variables: su valor se debe preservar entre los clics del botón, por lo que es esencial el alcance privado. Si eligiéramos el alcance local, se perdería su valor cada vez que el programa saliera del método **actionPerformed**.

El programa ofrece la posibilidad de crear animaciones. Si borramos el área de dibujo antes de volver a dibujar y hacemos que el programa vuelva a dibujar la imagen sin esperar a que hagamos clic, lograremos la ilusión de movimiento continuo.

## PRÁCTICA DE AUTOEVALUACIÓN

- 6.7** En el ejemplo **GotasDeLluvia**, la posición actual del deslizador se muestra en un campo de texto. ¿Cuáles serían las consecuencias de modificar la posición inicial del deslizador en la llamada de un constructor de **JSlider**?

## Fundamentos de programación

Durante muchos años el sueño de los programadores ha sido poder crear programas de la misma forma en que se construyen los sistemas de alta fidelidad; es decir, a partir de componentes comerciales tales como bocinas, amplificadores, controles de volumen, etc. Las mejoras en la programación orientada a objetos junto con las extensas bibliotecas de Java hacen que este sueño esté cada vez más cerca de cumplirse.

## Errores comunes de programación

- Si se declara una instancia pero se omite su inicialización con `new`, se producirá un error en tiempo de ejecución del tipo `nullPointerException`, al momento de tratar de utilizar dicha instancia. Los errores en tiempo de ejecución (o “bugs”) son más problemáticos que los errores en tiempo de compilación; son más difíciles de encontrar y son más graves, ya que se detiene la ejecución del programa.
- Los nombres de las clases de GUI de Java empiezan con `J`, como en el caso de `JButton`. Hay clases con nombres similares en la biblioteca AWT, pero sin la `J` (como `Button`). Se produce un error en tiempo de ejecución si utilizamos estas clases. Recuerde la `J`.

## Secretos de codificación

- Las variables de instancia se declaran fuera de los métodos, mediante `private`, como en el siguiente ejemplo:

```
private int suVariable;
private Random miVariable = new Random();
```

- Las variables de instancia se pueden inicializar al momento de su declaración o dentro de un método.

## Nuevos elementos del lenguaje

- Las variables de instancia `private`.
- El uso de `new` para crear instancias.
- La instrucción `import` para facilitar el uso de las bibliotecas.
- Las clases `JButton`, `JLabel`, `JTextField`, `Random`, `JSlider`, `ImageIcon` y `Timer`.

## Resumen

El sistema de Java tiene una gran cantidad de clases que podemos (y debemos) usar. No escriba su propio código sin antes investigar las bibliotecas.

## Ejercicios

- 6.1** Escriba un programa que calcule el área de un rectángulo. Las dimensiones se deben introducir mediante campos de texto y tiene que mostrar el resultado en un campo de texto. Asegúrese de que los campos de entrada estén identificados con claridad.

**6.2** Escriba un programa que produzca un número aleatorio entre 200 y 400 cada vez que se haga clic en un botón. El programa deberá mostrar este número, junto con la suma y el promedio de todos los números hasta ese momento. A medida que usted oprime el botón una y otra vez, el promedio deberá convergir hacia 300. Si no lo hace, entonces sospecharemos del generador de números aleatorios, ¡así como sospecharíamos de una moneda que al tirarla cayera en cara durante 100 veces seguidas!

**6.3** (a) Escriba un programa que convierta los grados Celsius en grados Fahrenheit. El usuario debe introducir el valor Celsius en un campo de texto; utilice valores enteros. Al hacer clic en un botón deberá aparecer el valor Fahrenheit en otro campo de texto. La fórmula de conversión es:

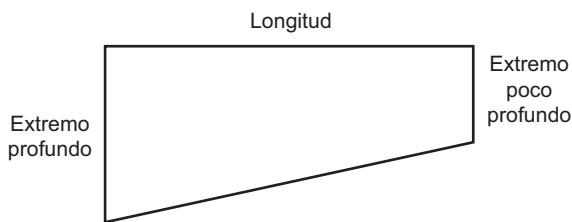
$$f = (c * 9) / 5 + 32;$$

- (b) Modifique el programa de manera que se introduzca el valor Celsius mediante un control deslizable, el cual deberá tener su valor mínimo en **0** y su valor máximo en **100**.
- (c) Represente ambas temperaturas como rectángulos largos y delgados que se dibujen después de cada evento de “cambio”. Recuerde borrar el área de dibujo y restablecer el color en cada ocasión.

**6.4** Escriba un programa que calcule el volumen de una alberca y muestre su sección transversal en un cuadro de imagen. La anchura de la alberca está fija en 5 metros y la longitud está fija en 20 metros. El programa debe tener dos controles deslizables: uno para ajustar la profundidad del extremo profundo y otro para ajustar la profundidad del extremo poco profundo. La profundidad mínima de cada extremo debe ser de 1 metro. Vuelva a dibujar la alberca en el método **stateChanged**. Seleccione valores apropiados para los valores mínimo y máximo del control deslizable en tiempo de diseño. La fórmula del volumen es:

$$v = \text{profundidadPromedio} * \text{anchura} * \text{longitud};$$

La figura 6.8 muestra la sección transversal de la alberca



**Figura 6.8** Sección transversal de una alberca.

**6.5** Escriba un programa que muestre los minutos y segundos cambiantes, representándolos mediante dos rectángulos largos: haga que la anchura máxima de los rectángulos sea de 600 píxeles para simplificar la aritmética (10 píxeles por cada minuto y cada segundo). Establezca el tamaño del marco en 700 píxeles de ancho y la anchura preferida del panel de dibujo en 700 píxeles. Vuelva a dibujar los dos rectángulos cada segundo. La figura 6.9 muestra una representación de 30 minutos y 15 segundos.

El programa debe contar en segundos con un temporizador y mostrar el total de segundos, además del tiempo en minutos y segundos. Para agilizar la prueba del programa reduzca el intervalo de tiempo de 1000 milisegundos a 200, por ejemplo.



600 píxeles de anchura

**Figura 6.9** Despliegue del tiempo para 30 minutos con 15 segundos.

**6.6** Esta pregunta lo llevará a través del proceso de escribir un juego de geometría:

- Escriba un programa con dos controles deslizables que controlen la posición horizontal y vertical de un círculo que tenga 200 píxeles de diámetro. Borre la pantalla y vuelva a dibujar el círculo en el método **stateChanged**.
- Agregue un tercer control deslizable para controlar el diámetro del círculo.
- Lo que sigue es un juego basado en el hecho matemático de que se puede dibujar un círculo en base a tres puntos cualesquiera. En un principio el programa debe mostrar tres puntos (cada uno de ellos es un pequeño círculo relleno). Algunas posiciones iniciales convenientes son (100, 100), (200, 200) y (200, 100), pero puede agregar un pequeño número aleatorio a estas posiciones para obtener más variedad. El jugador debe manipular el círculo hasta que éste pase por cada uno de los puntos.

## Respuestas a las prácticas de autoevaluación

**6.1** El programa se compilará y ejecutará de todas formas, pero es muy probable que produzca resultados incorrectos. Ahora modifica el valor de una variable que se puede usar en otros métodos. Antes modificaba una variable local.

**6.2** Cada clic del botón dibuja una línea vertical de 100 píxeles de largo. Cada línea se localiza 10 píxeles a la derecha de la anterior.

**6.3** La segunda instrucción se debe colocar dentro de un método. Como alternativa se podría usar la forma de una sola instrucción, como en el siguiente ejemplo:

```
private Random r = new Random();
```

**6.4** int edad = aleatorioEnRango(16, 59);

**6.5** Agregamos un campo de texto a la ventana mediante la misma codificación que en el programa **SumarCamposTexto**. Un nombre apropiado sería **campoConteo**. En vez de mostrar la respuesta en un cuadro de mensaje, usamos lo siguiente:

```
campoConteo.setText(Integer.toString(conteoAutos));
```

**6.6** Las variables locales se crean de nuevo al entrar a su método y sus valores se borran cuando el método termina. Si **tics** fuera local no se mantendría el conteo.

**6.7** No hay consecuencias, ya que el valor del campo de texto se inicializa con el valor actual del control deslizable, sin importar cuál sea.

# CAPÍTULO 7



## Selección

En este capítulo conoceremos cómo:

- Utilizar las instrucciones **if** y **switch** para llevar a cabo evaluaciones.
- Manejar múltiples eventos.
- Utilizar los operadores de comparación tales como **>**;
- Utilizar los operadores lógicos **&&**, **||** y **!**;
- Declarar y utilizar datos **boolean**.
- Comparar cadenas.

### ● Introducción

Todos hacemos selecciones en la vida diaria. Usamos un abrigo si llueve. Compramos un CD si tenemos suficiente dinero. Las selecciones también se utilizan mucho en los programas. La computadora evalúa un valor *y*, de acuerdo con el resultado, toma un curso de acción u otro. Cada vez que el programa tiene una selección de acciones y decide realizar una o la otra, se utiliza una instrucción **if** o **switch** para describir la situación.

Ya hemos visto que un programa computacional consiste en una serie de instrucciones para una computadora. Ésta obedece las instrucciones, una tras otra, en secuencia. Pero algunas veces requerimos que la computadora evalúe ciertos datos y después seleccione una de varias acciones dependiendo del resultado de la evaluación. Por ejemplo, tal vez necesitemos que la computadora evalúe la edad de alguien y después le diga que puede votar o que es demasiado joven. A esto se le conoce como selección y se utiliza una instrucción conocida como **if**, el tema central de este capítulo.

Las instrucciones **if** son tan importantes que se utilizan en todos los lenguajes de programación que se han inventado hasta este momento.

## ● La instrucción **if**

Nuestro primer ejemplo es un programa que simula el candado digital en una bóveda. En la figura 7.1 se muestra la pantalla. La bóveda está cerrada a menos que el usuario introduzca el código correcto en un campo de texto, que en un principio está vacío. Al hacer clic en el botón, el programa convierte el texto introducido en un número y lo compara con el código correcto. Si el código es correcto, se muestra un mensaje.



Figura 7.1 Pantalla del programa de la bóveda.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Bóveda extends JFrame implements ActionListener {

 private JLabel greetingLabel;
 private JTextField campoCódigo;
 private JButton botón;
 private JTextField campoTextoResultado;

 public static void main(String[] args) {
 Bóveda demo = new Bóveda();
 demo.setSize(100,150);
 demo.crearGUI();
 demo.setVisible(true);
 }

 private void crearGUI() {
 setDefaultCloseOperation(EXIT_ON_CLOSE);
 Container ventana = getContentPane();
 ventana.setLayout(new FlowLayout());

 etiquetaBienvenida = new JLabel("escriba el código");
 ventana.add(etiquetaBienvenida);
```

```

campoCódigo = new JTextField(5);
ventana.add(campoCódigo);

botón = new JButton("abrir");
ventana.add(botón);
botón.addActionListener(this);

campoTextoResultado = new JTextField(5);
ventana.add(campoTextoResultado);

}

public void actionPerformed(ActionEvent event) {
 String cadenaCódigo;
 int código;

 cadenaCódigo = campoCódigo.getText();
 código = Integer.parseInt(cadenaCódigo);
 if (código == 123) {
 campoTextoResultado.setText("abierta");
 }
}
}

```

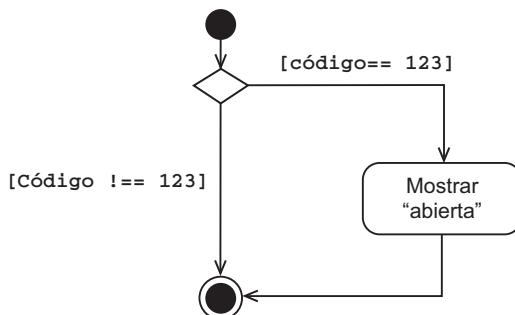


La instrucción **if** prueba el valor del número introducido. Si es igual al valor 123, se ejecuta la instrucción que está entre las llaves `{` y `}`. A continuación, se ejecuta cualquier instrucción que se encuentre después de la llave de cierre. Por otro lado, si el número no es igual a 123, se ignora la instrucción entre las llaves y se ejecuta cualquier instrucción que esté después de la llave de cierre.

Tenga en cuenta que la condición a evaluar se encierra entre paréntesis; ésta es una regla gramatical de Java.

Observe también que una prueba de igualdad utiliza el operador `==` (no el operador `=`, que se utiliza para asignación).

Una forma de visualizar a una instrucción **if** es mediante un diagrama de actividades (figura 7.2). Aquí se muestra la instrucción **if** anterior en forma gráfica. Para utilizar este diagrama, empiece en el círculo relleno de la parte superior y siga las flechas. Una decisión se muestra como un diamante,



**Figura 7.2** Diagrama de actividades de una instrucción **if**.

y las dos posibles condiciones se muestran entre corchetes. Las acciones se muestran dentro de cuadros con esquinas redondas y el fin de la secuencia es un círculo con una forma especial en la parte inferior del diagrama.

La instrucción **`if`** consta de dos partes:

- La condición que se va a evaluar.
- La instrucción o secuencia de instrucciones a ejecutar si la condición es verdadera.

Todos los programas consisten en una secuencia de acciones, y la secuencia evidente en el ejemplo anterior es:

1. Se recibe una pieza de texto del campo de texto.
2. Luego se realiza una evaluación.
3. Si es apropiado, se muestra un mensaje para indicar que la bóveda está abierta.

Es muy frecuente que se lleven a cabo no sólo una acción sino una secuencia completa de acciones si el resultado de la evaluación es verdadero; estas acciones deben ir encerradas entre las llaves.

Cabe mencionar que se aplica sangría a una línea para reflejar la estructura de la instrucción **`if`**. Sangría significa que se utilizan espacios para desplazar el texto hacia la derecha. Aunque no es esencial el uso de sangrías, es muy conveniente para que el lector (humano) de un programa lo pueda entender con facilidad. Todos los programas bien hechos (cuálquiera que sea el lenguaje) tienen sangría y todos los programadores con experiencia la utilizan.

## PRÁCTICA DE AUTOEVALUACIÓN

- 7.1 Mejore la instrucción **`if`** de manera que borre el número cuando se introduzca el código correcto en el campo de texto.

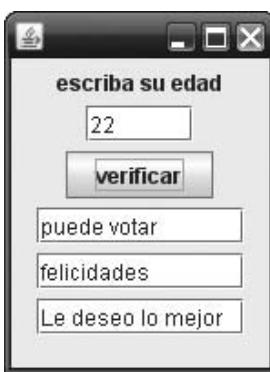
### ● **`if...else`**

Algunas veces es necesario especificar dos secuencias de acciones: las que se llevan a cabo si la condición es verdadera y las que se ejecutan si la condición es falsa.

Para verificar su capacidad de votar, el usuario del programa introduce su edad en un campo de texto y el programa decide si puede votar o no. La pantalla se muestra en la figura 7.3. Cuando el usuario hace clic en el botón, el programa extrae la información que el usuario introdujo en el campo de texto, convierte la cadena en un entero y coloca el número en la variable llamada **`edad`**. Después necesitamos que el programa realice diferentes acciones, dependiendo de si el valor es:

- Mayor que 17.
- Igual o menor que 17.

Para ello se utiliza la instrucción **`if`**:



**Figura 7.3** La pantalla del programa para verificar la capacidad de votar.

```

if (edad > 17) {
 campoDecisión.setText("puede votar");
 campoComentario.setText("felicitaciones");
}
else {
 campoDecisión.setText("no puede votar");
 campoComentario.setText = ("lo siento");
}

```

Los resultados de la prueba se muestran en varios campos de texto. El programa completo es el siguiente:

```

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Votación extends JFrame implements ActionListener {

 private JLabel etiquetaBienvenida;
 private JTextField campoEdad;
 private JButton botón;
 private JTextField campoDecisión;
 private JTextField campoComentario;
 private JTextField campoDespedida;

 public static void main(String[] args) {
 Votación demo = new Votación();
 demo.setSize(150,200);
 demo.crearGUI();
 demo.setVisible(true);
 }
}

```

```
private void crearGUI() {
 setDefaultCloseOperation(EXIT_ON_CLOSE);
 Container ventana = getContentPane();
 ventana.setLayout(new FlowLayout());

 etiquetaBienvenida = new JLabel("escriba su edad");
 ventana.add(etiquetaBienvenida);

 campoEdad = new JTextField(5);
 ventana.add(campoEdad);

 botón = new JButton("verificar");
 ventana.add(botón);
 botón.addActionListener(this);

 campoDecisión = new JTextField(10);
 ventana.add(campoDecisión);

 campoComentario = new JTextField(10);
 ventana.add(campoComentario);

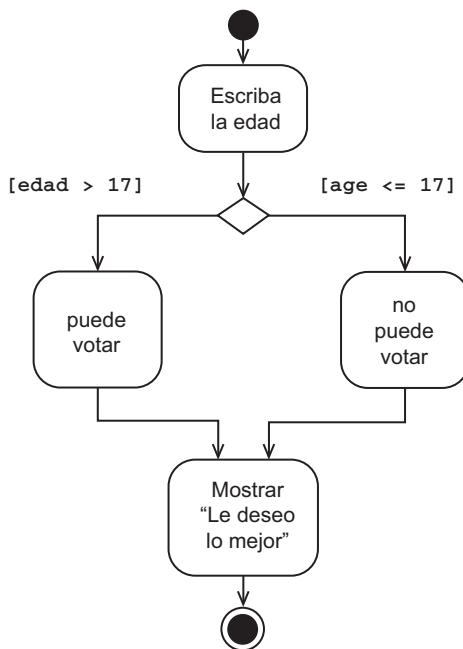
 campoDespedida = new JTextField(10);
 ventana.add(campoDespedida);
}

public void actionPerformed(ActionEvent event) {
 int edad;

 edad = Integer.parseInt(campoEdad.getText());
 if (edad > 17)
 {
 campoDecisión.setText("puede votar");
 campoComentario.setText("felicidades");
 }
 else
 {
 campoDecisión.setText("no puede votar");
 campoComentario.setText("lo siento");
 }
 campoDespedida.setText("Le deseo lo mejor");
}
```

En este programa la instrucción **if** consta de tres partes:

- La condición que se va a evaluar; en este caso, si la edad es mayor que 17.
- La instrucción o secuencia de instrucciones que se ejecutarán si la condición es verdadera, y que deben ir encerradas entre llaves.
- La instrucción o instrucciones a ejecutar si la condición es falsa, y que deben ir encerradas entre llaves.



**Figura 7.4** Diagrama de actividades para una instrucción `if...else`.

El nuevo elemento aquí es la palabra clave `else`, la cual introduce la segunda parte de la instrucción `if`. Observe de nuevo cómo ayuda la sangría a enfatizar la intención del programa.

Podemos visualizar una instrucción `if...else` como un diagrama de actividades, como se muestra en la figura 7.4. Este diagrama muestra la condición que se va a evaluar y las dos acciones separadas.

## Operadores de comparación

Los programas anteriores utilizan algunos de los operadores (algunas veces conocidos como relacionales) de comparación. He aquí una lista completa de ellos:

| Símbolo            | Significado       |
|--------------------|-------------------|
| <code>&gt;</code>  | mayor que         |
| <code>&lt;</code>  | menor que         |
| <code>==</code>    | igual que         |
| <code>!=</code>    | no es igual que   |
| <code>&lt;=</code> | menor o igual que |
| <code>&gt;=</code> | mayor o igual que |

Aquí podemos ver de nuevo que Java utiliza dos veces el signo “igual que” (`==`) para evaluar si dos cosas son iguales.

Siempre debe tener mucho cuidado al elegir el operador apropiado. En el programa para evaluar si alguien puede votar, probablemente la evaluación apropiada sería:

```
if (edad >= 18) {
 campoDecisión.setText("puede votar");
}
```

Tenga en cuenta que por lo general es posible escribir condiciones en una de dos formas. Los siguientes dos fragmentos de programa obtienen exactamente el mismo resultado, pero utilizan distintas condiciones:

```
if (edad >= 18) {
 campoDecisión.setText("puede votar");
}
else {
 campoDecisión.setText("lo siento");
}
```

obtiene el mismo resultado que:

```
if (edad < 18) {
 campoDecisión.setText("lo siento");
}
else {
 campoDecisión.setText("puede votar");
}
```

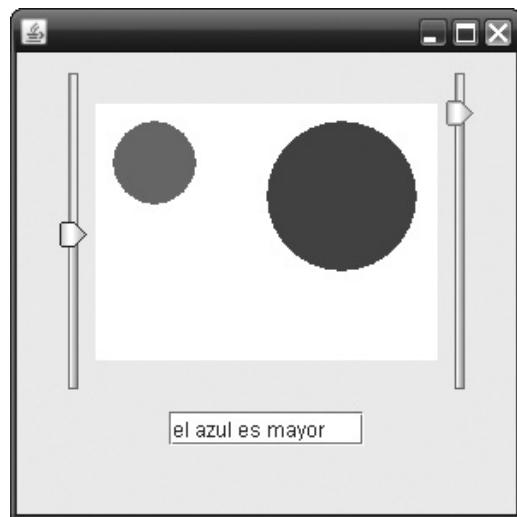
Aunque estos dos fragmentos logren el mismo resultado final, probablemente el primero sea mejor debido a que establece con más claridad la condición de poder votar.

## PRÁCTICA DE AUTOEVALUACIÓN

**7.2** ¿Obtienen las siguientes dos instrucciones `if` el mismo resultado o no?

```
if (edad > 18) {
 campoDecisión.setText("puede votar");
}
if (edad < 18) {
 campoDecisión.setText("no puede votar");
}
```

Algunas veces los humanos tienen dificultad para juzgar los tamaños relativos de círculos de distintos colores. El siguiente programa utiliza dos controles deslizables y muestra círculos con



**Figura 7.5** El programa mayor.

tamaños equivalentes (figura 7.5). Además compara los tamaños e informa cuál es el mayor. Primero vamos a escribir el programa mediante la siguiente instrucción **if**:

```
if (valorRojo > valorAzul) {
 campoTexto.setText("el rojo es mayor");
}
else {
 campoTexto.setText("el azul es mayor");
}
```

El método de biblioteca **fillOval** se utiliza para dibujar un círculo relleno cuyo diámetro sea igual al valor obtenido del control deslizable correspondiente. El programa completo es:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;

public class Mayor extends JFrame
 implements ChangeListener {

 private JSlider deslizanteRojo;
 private JPanel panel;
 private JSlider deslizanteAzul;
 private JTextField campoTexto;
```

```
public static void main(String[] args) {
 Mayor demo = new Mayor();
 demo.setSize(300,300);
 demo.crearGUI();
 demo.setVisible(true);
}

private void crearGUI() {
 setDefaultCloseOperation(EXIT_ON_CLOSE);
 Container ventana = getContentPane();
 ventana.setLayout(new FlowLayout());

 deslizanteRojo = new JSlider(JSlider.VERTICAL);
 deslizanteRojo.addChangeListener(this);
 ventana.add(deslizanteRojo);

 panel = new JPanel();
 panel.setPreferredSize(new Dimension(200, 150));
 panel.setBackground(Color.white);
 ventana.add(panel);

 deslizanteAzul = new JSlider(JSlider.VERTICAL);
 deslizanteAzul.addChangeListener(this);
 ventana.add(deslizanteAzul);

 campoTexto = new JTextField(10);
 ventana.add(campoTexto);
}

public void stateChanged(ChangeEvent e) {
 Graphics papel = panel.getGraphics();

 int valorRojo, valorAzul;
 valorRojo = deslizanteRojo.getValue();
 valorAzul = deslizanteAzul.getValue();

 papel.setColor(Color.white);
 papel.fillRect(0, 0, 200, 150);
 papel.setColor(Color.red);
 papel.fillOval(10, 10, valorRojo, valorRojo);
 papel.setColor(Color.blue);
 papel.fillOval(100, 10, valorAzul, valorAzul);

 if (valorRojo > valorAzul) {
 campoTexto.setText("el rojo es mayor");
 }
 else {
 campoTexto.setText("el azul es mayor");
 }
}
```

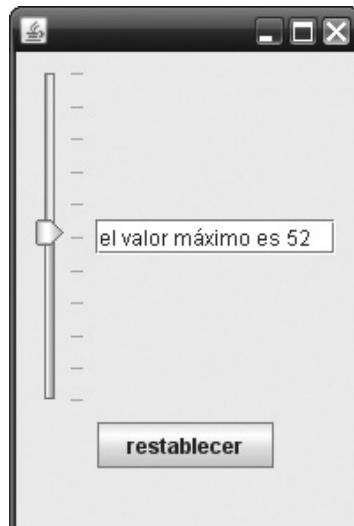
Este programa funciona bien, sólo que ilustra de nuevo la importancia de tener cuidado al usar instrucciones `if`. En este programa, ¿qué ocurre cuando los dos valores son iguales? La respuesta es que el programa encuentra que el azul es mayor; lo que en definitiva no es así. Podríamos mejorar el programa para establecer las cosas con más claridad cambiando la instrucción `if` por el siguiente código:

```
if (valorRojo > valorAzul) {
 campoTexto.setText("el rojo es mayor");
}
if (valorAzul > valorRojo) {
 campoTexto.setText("el azul es mayor");
}
if (valorRojo == valorAzul) {
 campoTexto.setText("Son iguales");
}
```

El siguiente ejemplo es un programa que lleva el registro del valor más grande de un número, a medida que va cambiando. Es un ejercicio común en programación. Algunos termómetros tienen un mecanismo para registrar la temperatura máxima alcanzada. Este programa simula dicho termómetro mediante un control deslizable. Muestra el máximo valor al que está ajustado el control deslizable (vea la figura 7.6).

El programa utiliza una variable llamada `max`, una variable a nivel de clase que contiene el valor de la temperatura más alta alcanzada hasta ese momento. La variable `max` se declara así:

```
private int max = 0;
```



**Figura 7.6** Pantalla del termómetro.

Una instrucción **if** compara el valor actual del control deslizable con el valor de **max** y lo modifica si es necesario:

```
int temp;

temp = deslizante.getValue();
if (temp > max) {
 max = temp;
}
```

He aquí el programa completo:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;

public class Max extends JFrame implements ChangeListener,
 ActionListener {

 private JSlider deslizante;
 private JTextField campoTexto;
 private JButton botón;

 private int max = 0;

 public static void main(String[] args) {
 Max demo = new Max();
 demo.setSize(200,300);
 demo.crearGUI();
 demo.setVisible(true);
 }

 private void crearGUI() {
 setDefaultCloseOperation(EXIT_ON_CLOSE);
 Container ventana = getContentPane();
 ventana.setLayout(new FlowLayout());
 deslizante = new JSlider(JSlider.VERTICAL, 0, 100, 0);
 deslizante.setMajorTickSpacing(10);
 deslizante.setPaintTicks(true);
 deslizante.addChangeListener(this);
 ventana.add(deslizante);

 campoTexto = new JTextField(12);
 ventana.add(campoTexto);

 botón = new JButton("restablecer");
 botón.addActionListener(this);
 ventana.add(botón);
 }
}
```

```

public void stateChanged(ChangeEvent e) {
 int temp;
 temp = deslizante.getValue();
 if (temp > max) {
 max = temp;
 }
 mostrar();
}

public void actionPerformed(ActionEvent event) {
 campoTexto.setText("");
 max = 0;
}

private void mostrar() {
 campoTexto.setText("el valor máximo es " + max);
}

```

## PRÁCTICA DE AUTOEVALUACIÓN

- 7.3** Escriba un programa que muestre el valor numérico del valor mínimo al que se ajuste el control deslizable.

Ahora veremos un programa que simula la acción de tirar dos dados. La computadora decide los valores de los dados al azar. Debemos crear un botón con la leyenda “tirar”. Al hacer clic en este botón, el programa obtendrá dos números al azar y los utilizará como los valores de los dados (figura 7.7). El programa usa una instrucción **if** para decidir si los valores son iguales.

Para obtener un número aleatorio usamos el método **nextInt** de la clase de biblioteca **Random**. Vimos esta clase en el capítulo 6. Lo que necesitamos para nuestro propósito es un entero en el rango de 1 a 6, por lo que obtenemos números en el rango de 0 a 5 y simplemente les sumamos 1.



Figura 7.7 Apuestas.

El programa que simula tirar dos dados es el siguiente:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.util.*;

public class Apuestas extends JFrame implements ActionListener {

 private JButton botón;
 private JTextField campoTextoValores, campoTextoResultado;
 private Random aleatorio;

 public static void main(String[] args) {
 Apuestas demo = new Apuestas();
 demo.setSize(200,150);
 demo.crearGUI();
 demo.setVisible(true);
 }

 private void crearGUI() {
 setDefaultCloseOperation(EXIT_ON_CLOSE);
 Container ventana = getContentPane();
 ventana.setLayout(new FlowLayout());

 botón = new JButton("tirar");
 ventana.add(botón);
 botón.addActionListener(this);

 campoTextoValores = new JTextField(14);
 ventana.add(campoTextoValores);

 campoTextoResultado = new JTextField(14);
 ventana.add(campoTextoResultado);

 aleatorio = new Random();
 }

 public void actionPerformed(ActionEvent event) {
 int dado1, dado2;
 dado1 = aleatorio.nextInt(6) + 1;
 dado2 = aleatorio.nextInt(6) + 1;

 campoTextoValores.setText("valores de los dados: "
 + Integer.toString(dado1) + " y "
 + Integer.toString(dado2));
 if (dado1 == dado2) {
 campoTextoResultado.setText("dados iguales - gana");
 }
 else {
 campoTextoResultado.setText("dados no iguales - pierde");
 }
 }
}
```

## Múltiples eventos

La instrucción `if` puede cumplir un importante papel en el manejo de múltiples eventos. Por ejemplo, un programa tiene dos botones con las leyendas feliz y triste, como se muestra en la figura 7.8. Al hacer clic en un botón, el programa muestra una imagen apropiada. Sin importar en cuál de los botones se haga clic, se hace una llamada al mismo método `actionPerformed`. Entonces, ¿cómo podemos diferenciar uno del otro? Se utiliza una instrucción `if` para detectar cuál fue el botón que se oprimió. Después se puede llevar a cabo la acción correspondiente.

Cuando ocurre un evento de botón, el sistema operativo llama al método `actionPerformed`. Esta parte, crucial, del programa es:

```
public void actionPerformed(ActionEvent event) {
 Graphics papel = panel.getGraphics();
 Object origen = event.getSource();
 if (origen == botónFeliz) {
 imagenFeliz.paintIcon(this, papel, 0, 0);
 }
 else {
 imagenTriste.paintIcon(this, papel, 0, 0);
 }
}
```

El parámetro que se pasa a `actionPerformed`, de nombre `event` en el programa, provee información sobre la naturaleza del evento. Al usar el método `getSource` en `event` se devuelve el objeto responsable del evento. Colocamos esto en una variable llamada `origen`, un objeto de la clase `Object`. Despues utilizamos una instrucción `if` para comparar este objeto con los posibles candidatos. Así, utilizamos el operador `==` para comparar objetos, no números o cadenas (tenga en cuenta que no es el operador `=`).



Figura 7.8 Múltiples botones.

He aquí el texto completo del programa:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class FelizOTriste extends JFrame implements ActionListener {

 private JButton botónFeliz, botónTriste;
 private JPanel panel;

 private ImageIcon imagenFeliz, imagenTriste;

 public static void main(String[] args) {
 FelizOTriste demo = new FelizOTriste();
 demo.setSize(175,175);
 demo.crearGUI();
 demo.setVisible(true);
 }

 private void crearGUI() {
 setDefaultCloseOperation(EXIT_ON_CLOSE);
 Container ventana = getContentPane();
 ventana.setLayout(new FlowLayout());

 panel = new JPanel();
 panel.setPreferredSize(new Dimension(100, 100));
 panel.setBackground(Color.white);
 ventana.add(panel);

 botónFeliz = new JButton("feliz");
 ventana.add(botónFeliz);
 botónFeliz.addActionListener(this);

 botónTriste= new JButton("triste");
 ventana.add(botónTriste);
 botónTriste.addActionListener(this);

 imagenFeliz = new ImageIcon("feliz.jpg");
 imagenTriste = new ImageIcon("triste.jpg");
 }

 public void actionPerformed(ActionEvent event) {
 Graphics papel = panel.getGraphics();
 Object origen = event.getSource();
 if (origen == botónFeliz) {
 imagenFeliz.paintIcon(this, papel, 0, 0);
 }
 else {
 imagenTriste.paintIcon(this, papel, 0, 0);
 }
 }
}
```

## PRÁCTICA DE AUTOEVALUACIÓN

- 7.4** Un programa muestra tres botones, con las leyendas 1, 2 y 3. Escriba las instrucciones para mostrar el número del botón en el que se hizo clic.

### ● And, or, not

A menudo en programación necesitamos evaluar dos cosas a la vez. Suponga, por ejemplo, que necesitamos evaluar si alguien debe pagar una tarifa reducida por un boleto:

```
if (edad >= 6 && edad < 16) {
 campoTexto.setText("tarifa infantil");
}
```

La palabra `&&` es uno de los operadores lógicos de Java; significa “and” en inglés e “y” en español común.

Se pueden utilizar paréntesis adicionales para mejorar la legibilidad de estas condiciones más complejas. Por ejemplo, podemos replantear la instrucción anterior de la siguiente manera:

```
if ((edad >= 6) && (edad < 16)) {
 campoTexto.setText("tarifa infantil");
}
```

Aunque los paréntesis internos no son esenciales, sirven para diferenciar las dos condiciones que se están evaluando.

Podría verse tentado a escribir:

```
if (edad >= 6 && < 16) // ¡error!
```

pero es incorrecto, ya que las condiciones se tienen que establecer por completo, como se muestra a continuación:

```
if (edad >= 6 && edad < 16) // OK
```

Podríamos utilizar el operador `||` (que significa “or” en inglés y “o” en español) en una instrucción `if` de la siguiente forma:

```
if (edad < 6 || edad >= 60) {
 campoTexto.setText("tarifa reducida");
}
```

aquí se aplica la tarifa reducida cuando las personas son menores de 6 o de 60 en adelante.

El operador `!` significa “not” en inglés (“no” en español) y se utiliza mucho en la programación. He aquí un ejemplo del uso del operador `!`:

```
if (! (edad > 16)) {
 campoTexto.setText("demasiado joven");
}
```

Esto significa: evaluar si la edad es mayor de 16. Si el resultado es verdadero, el `!` lo convierte en falso. Si es falso, el `!` lo convierte en verdadero. Entonces, si el resultado es verdadero, mostrar el mensaje. Desde luego que esto se puede escribir de una manera más sencilla sin el operador `!`.

## PRÁCTICA DE AUTOEVALUACIÓN

7.5 Rediseñe la instrucción `if` anterior sin utilizar el operador `!`.



**Figura 7.9** El programa de los dados.

El siguiente programa ilustra una serie más compleja de evaluaciones. Se lanzan dos dados en un juego de apuestas y el programa tiene que decidir cuál es el resultado. El programa usa dos controles deslizables, cada uno con un rango de 1 a 6 para especificar los valores de cada uno de los dos dados (figura 7.9). El resultado se muestra en dos campos de texto. Para empezar usaremos la regla que establece que sólo una puntuación total de 6 gana.

El programa se muestra a continuación. Cada vez que se mueve uno de los dos controles deslizables, el programa muestra el valor total y utiliza una instrucción `if` para determinar si el jugador ganó.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;

public class Dados extends JFrame implements ChangeListener {
 private JSlider deslizante1, deslizante2;
 private JTextField campoTextoTotal, campoTextoComentario;
 public static void main(String[] args) {
 Dados demo = new Dados();
 demo.setSize(200,150);
 demo.crearGUI();
 demo.setVisible(true);
 }
 private void crearGUI() {
 setDefaultCloseOperation(EXIT_ON_CLOSE);
 Container ventana = getContentPane();
 ventana.setLayout(new FlowLayout());
 deslizante1 = new JSlider(1, 6, 3);
 deslizante1.addChangeListener(this);
 ventana.add(deslizante1);
```

```

deslizante2 = new JSlider(1, 6, 3);
deslizante2.addChangeListener(this);
ventana.add(deslizante2);

campoTextoTotal = new JTextField(10);
ventana.add(campoTextoTotal);

campoTextoComentario= new JTextField(10);
ventana.add(campoTextoComentario);
}

public void stateChanged(ChangeEvent e) {
 int dado1, dado2, total;

 dado1 = deslizante1.getValue();
 dado2 = deslizante2.getValue();
 total = dado1 + dado2;
 campoTextoTotal.setText("el total es " + total);
 if (total == 6) {
 campoTextoComentario.setText("usted ha ganado");
 }
 else {
 campoTextoComentario.setText("usted ha perdido");
 }
}
}

```



Ahora vamos a modificar las reglas y a rediseñar el programa. Suponga que cualquier par de valores idénticos gana; por ejemplo, los dos dados con uno, con dos, etc. Entonces la instrucción **if** sería:

```

if (dado1 == dado2) {
 campoTextoComentario.setText("usted ha ganado");
}

```

Ahora supongamos que usted gana sólo gana si obtiene un total de 2 o 7:

```

if ((total == 2) || (total == 7)) {
 campoTextoComentario.setText("usted ha ganado");
}

```

Observe de nuevo que hemos encerrado cada una de las condiciones entre paréntesis. Estos paréntesis no son estrictamente necesarios en Java, pero ayudan mucho para aclarar el significado de la condición a evaluar.

En la siguiente tabla se sintetizan los operadores **and**, **or** y **not** de Java:

| Símbolo | Significado |
|---------|-------------|
| &&      | and         |
|         | or          |
| !       | not         |

## PRÁCTICAS DE AUTOEVALUACIÓN

- 7.6 Modifique el programa de los dados para que el jugador gane cuando obtenga un valor total de 2, 5 o 7.
- 7.7 Escriba instrucciones ***if*** para evaluar si alguien puede obtener un empleo de tiempo completo. La regla es que debe tener 16 o más años y ser menor que 65.

### Instrucciones ***if*** anidadas

Analice el siguiente fragmento de un programa:

```
if (edad < 6) {
 campoTexto.setText("tarifa infantil");
}
else {
 if (edad < 16) {
 campoTexto.setText("tarifa junior");
 }
 else {
 campoTexto.setText("tarifa de adulto");
 }
}
```

Aquí podemos ver que la segunda instrucción ***if*** está completamente dentro de la primera (la sangría ayuda a mejorar la legibilidad). A esto se le conoce como anidamiento. El anidamiento no es lo mismo que aplicar sangría al código; es sólo que la sangría hace el anidamiento muy evidente.

El efecto general de esta pieza de código es:

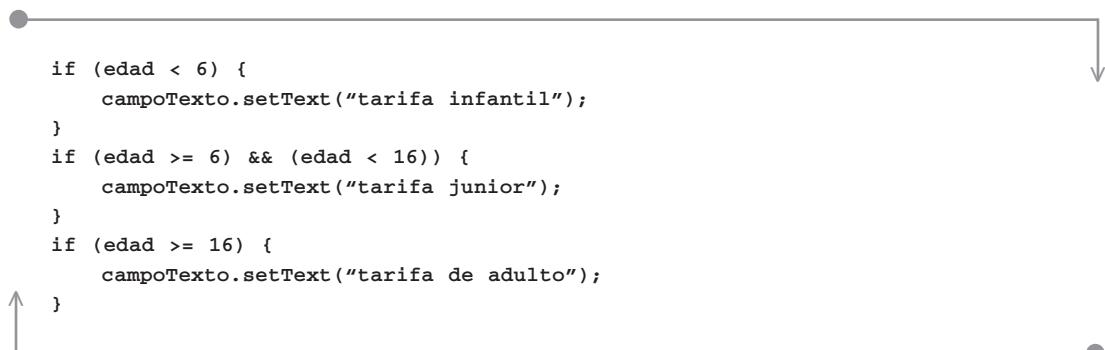
- Si la edad es mayor que 6, se aplica la tarifa infantil.
- De otra manera, si la edad es menor que 16, se utiliza la tarifa junior.
- De otra manera, si ninguna de las anteriores condiciones es verdadera, se utiliza la tarifa de adulto.

Es común ver el anidamiento en los programas, pero un programa de este tipo tiene una complejidad que dificulta un poco su comprensión. Podemos rediseñar esta sección del programa en un estilo de instrucciones ***if*** anidadas conocidas como instrucciones ***else if***, como se muestra en el siguiente ejemplo:

```
if (edad < 6) {
 campoTexto.setText("tarifa infantil");
}
else if (edad < 16) {
 campoTexto.setText("tarifa junior");
}
else {
 campoTexto.setText("tarifa de adulto");
}
```

Esta versión es exactamente la misma que la anterior, sólo que la sangría es distinta y algunos de los pares de llaves se eliminaron. Esto se debe a que la regla establece que cuando hay una **sólo** instrucción a ejecutar, podemos omitir las llaves. Ésta es la única ocasión en que recomendamos omitir las llaves.

He aquí una tercera versión de esta pieza de programa. A menudo es posible escribir un programa de una manera más simple utilizando los operadores lógicos. Por ejemplo, en el siguiente ejemplo se obtiene el mismo resultado que en el ejemplo anterior sin utilizar el anidamiento:



Ahora tenemos tres piezas de código que obtienen el mismo resultado final, dos con anidamiento y uno sin esta técnica. Algunas personas argumentan que es difícil entender el anidamiento, de tal forma que el programa es propenso a errores y por ende es mejor evitar el anidamiento. A menudo es posible evitarlo si utilizamos operadores lógicos.

## PRÁCTICAS DE AUTOEVALUACIÓN

- 7.8** Escriba un programa que reciba un salario mediante un control deslizable y determine cuánto impuesto debe pagar alguien de acuerdo con las siguientes reglas:

Las personas no deben pagar impuesto si ganan hasta \$10,000. Deben pagar el 20% de impuesto si ganan más de \$10,000 y hasta \$50,000. Deben pagar el 90% de impuesto si ganan más de \$50,000. El control deslizable debe tener un rango de 0 a 100,000.

- 7.9** Escriba un programa que genere tres controles deslizables y muestre el mayor de los tres valores.

- 7.10** La agencia de viajes Joven y Bella sólo acepta clientes entre los 18 y 30 años (si es menor de 18 no tiene dinero; si es mayor de 30 tiene demasiadas arrugas). Escriba un programa para evaluar si usted puede irse de vacaciones con esta empresa. La edad se introduce en un campo de texto. El resultado se muestra en un segundo campo de texto cuando se hace clic en un botón.

## La instrucción switch

Esta instrucción es otra forma de usar muchas instrucciones **if**. Siempre podrá realizar todo lo que necesite con la ayuda de las instrucciones **if**, pero **switch** puede ser una alternativa más eficiente en circunstancias apropiadas. Por ejemplo, suponga que necesitamos una pieza de código para mostrar el día de la semana como una cadena de texto. Suponga que el programa representa el día de la semana como una variable **int** llamada **númeroDía**, que tiene uno de los valores del 1 al 7 para representar los días de lunes a domingo. Queremos convertir la versión entera del día en una versión de cadena de texto llamada **nombreDía**. Podríamos escribir la siguiente serie de instrucciones **if**:

```
if (númeroDía == 1) {
 nombreDía = "Lunes";
}
if (númeroDía == 2) {
 nombreDía = "Martes";
}
if (númeroDía == 3) {
 nombreDía = "Miércoles";
}
if (númeroDía == 4) {
 nombreDía = "Jueves";
}
if (númeroDía == 5) {
 nombreDía = "Viernes";
}
if (númeroDía == 6) {
 nombreDía = "Sábado";
}
if (númeroDía == 7) {
 nombreDía = "Domingo";
}
```

Aunque la anterior pieza de código está clara y bien estructurada, hay una alternativa que cumple la misma función, en la cual se utiliza la instrucción **switch**:

```
switch (númeroDía) {

 case 1:
 nombreDía = "Lunes";
 break;

 case 2:
 nombreDía = "Martes";
 break;
```

```

case 3:
 nombreDía = "Miércoles";
 break;

case 4:
 nombreDía = "Jueves";
 break;

case 5:
 nombreDía = "Viernes";
 break;

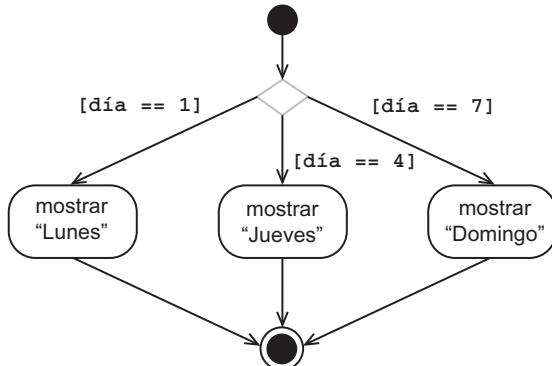
case 6:
 nombreDía = "Sábado";
 break;

case 7:
 nombreDía = "Domingo";
 break;
}

```

Se antepone la palabra **case** a cada uno de los posibles valores. La instrucción **break** transfiere el control al final de la instrucción **switch**, el cual está marcado con una llave de cierre. Esto nos permite ver con más claridad lo que se va a hacer, en contraste con la serie equivalente de instrucciones **if**.

Una instrucción **switch** como la anterior se puede visualizar como el diagrama de actividades de la figura 7.10.



**Figura 7.10** Diagrama de actividades que muestra parte de una instrucción **switch**.

## PRÁCTICA DE AUTOEVALUACIÓN

- 7.11** Escriba un método que convierta los enteros 1, 2, 3 y 4 en las palabras diamantes, corazones, tréboles y espadas, respectivamente.

Puede haber varias instrucciones en cada una de las opciones de una instrucción **switch**. Por ejemplo, una de las opciones podría ser:

```
case 6:
 JOptionPane.showMessageDialog(null, "hurra");
 nombreDía = "Sábado";
 break;
```

Otra característica de la instrucción **switch** es la de agrupar varias opciones, como en el siguiente ejemplo:

```
switch (númeroDía) {
 case 1:
 case 2:
 case 3:
 case 4:
 case 5:
 nombreDía = "entre semana";
 break;

 case 6:
 case 7:
 nombreDía = "fin de semana"
 break;
}
```

The diagram shows a horizontal line with two arrows pointing downwards from the start and end of the switch block. From each arrow, a vertical line extends downwards to the start and end of the entire switch statement respectively.

La opción **default** es otra parte de la instrucción **switch** que sirve para ciertos casos. Suponga en el ejemplo anterior que el valor del entero que denota el día de la semana se introduce mediante un cuadro de texto. Entonces existe la clara posibilidad de que el usuario introduzca en forma errónea un número que no se encuentre en el rango de 1 a 7. Cualquier programa decente necesita tener esto en cuenta para evitar que ocurra algo extraño, o que el programa falle. La instrucción **switch** es muy buena para lidiar con esta situación, ya que podemos proveer una opción “atrapa todo” o predeterminada, la cual se utiliza si ninguna de las otras opciones es válida:

```
switch (númeroDía) {

 case 1:
 nombreDía = "Lunes";
 break;

 case 2:
 nombreDía = "Martes";
 break;

 case 3:
 nombreDía = "Miércoles";
 break;
}
```

The diagram shows a horizontal line with two arrows pointing downwards from the start and end of the switch block. From each arrow, a vertical line extends downwards to the start and end of the entire switch statement respectively.

```
case 4:
 nombreDía = "Jueves";
 break;

case 5:
 nombreDía = "Viernes";
 break;

case 6:
 nombreDía = "Sábado";
 break;

case 7:
 nombreDía = "Domingo";
 break;

default:
 nombreDía = "día ilegal";
 break;
}
```

Si no se escribe una opción **default** como parte de una instrucción **switch** y ninguno de los casos provistos corresponde con el valor actual de la variable, entonces se ignoran todas las opciones.

## Variables booleanas

Todos los tipos de variables que hemos visto hasta ahora están diseñados para contener números, cadenas de texto u objetos. Ahora veremos un nuevo tipo de variable llamada **boolean**, la cual sólo puede contener el valor **true** (verdadero) o el valor **false** (falso). Las palabras **boolean**, **true** y **false** son palabras reservadas en Java, por lo cual no se pueden utilizar para ningún otro fin. Este tipo de variable recibió su nombre en honor al matemático inglés del siglo diecinueve George Boole, quien hizo una gran contribución al desarrollo de la lógica matemática, en donde las ideas de verdadero y falso desempeñan un papel central.

El siguiente programa muestra un anuncio en una tienda (figura 7.11). El anuncio dice abierta o cerrada. Se utiliza una variable **boolean** llamada **abierta** para registrar si la tienda está abierta (**true**) o cerrada (**false**). Hay dos botones que permiten al encargado de la tienda cambiar el anuncio a abierta o cerrada. Hay otros dos botones que prenden y apagan el anuncio. El programa muestra letras de tamaño grande mediante el método **setFont**.

La variable **boolean abierta** es una variable a nivel de clase, que al principio es falsa para indicar que la tienda está cerrada:

```
private boolean abierta = false;
```

Al hacer clic en el botón **Abierta**:

```
abierta = true;
```

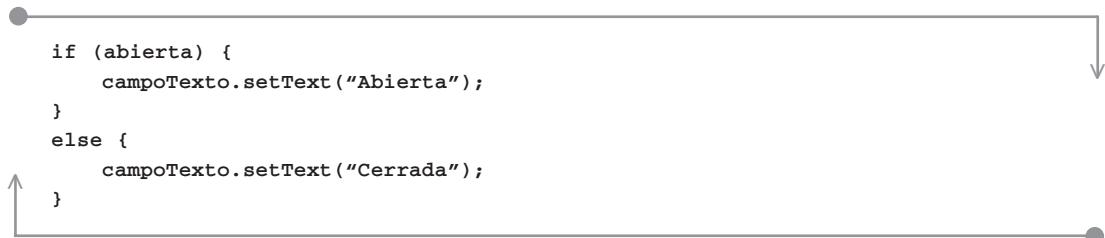


Figura 7.11 El anuncio de una tienda.

Al hacer clic en el botón **Cerrada**:

```
abierta = false;
```

Al hacer clic en el botón **Prendido**, el valor de **abierta** se evalúa mediante una instrucción **if** y se muestra el anuncio apropiado:



He aquí el programa completo:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class AnuncioTienda extends JFrame implements ActionListener {

 private JButton botónPrendido, botónApagado, botónAbierta, botónCerrada;
 private JTextField campoTexto;

 private boolean prendido = false, abierta = false;
 public static void main(String[] args) {
 AnuncioTienda demo = new AnuncioTienda();
 demo.setSize(300,200);
 demo.crearGUI();
 demo.setVisible(true);
 }
}
```

```
private void crearGUI() {
 setDefaultCloseOperation(EXIT_ON_CLOSE);
 Container ventana = getContentPane();
 ventana.setLayout(new FlowLayout());
 botónPrendido = new JButton("Prendido");
 ventana.add(botónPrendido);
 botónPrendido.addActionListener(this);
 botónApagado = new JButton("Apagado");
 ventana.add(botónApagado);
 botónApagado.addActionListener(this);
 campoTexto = new JTextField(5);
 campoTexto.setSize(5, 100);
 campoTexto.setFont(new Font(null, Font.BOLD, 60));
 ventana.add(campoTexto);
 botónAbierta = new JButton("Abierta");
 ventana.add(botónAbierta);
 botónAbierta.addActionListener(this);
 botónCerrada = new JButton("Cerrada");
 ventana.add(botónCerrada);
 botónCerrada.addActionListener(this);
}
public void actionPerformed(ActionEvent event) {
 Object origen = event.getSource();
 if (origen == botónPrendido) {
 manejarBotónPrendido();
 }
 else if (origen == botónApagado) {
 manejarBotónApagado();
 }
 else if (origen == botónAbierta) {
 manejarBotónAbierta();
 }
 else manejarBotónCerrada();
 dibujarAnuncio();
}
private void manejarBotónPrendido() {
 prendido = true;
}
private void manejarBotónApagado() {
 prendido = false;
}
private void manejarBotónAbierta() {
 abierta = true;
}
private void manejarBotónCerrada() {
 abierta = false;
}
```

```

private void dibujarAnuncio() {
 if (abierta) {
 campoTexto.setText("Abierta");
 }
 else {
 campoTexto.setText("Cerrada");
 }
 if (!prendido) {
 campoTexto.setText("");
 }
}

```



En el programa anterior, una de las instrucciones `if` es como se muestra a continuación, ya que la variable `abierta` puede ser verdadera o falsa, y se puede evaluar en forma directa:

```
if (abierta) {
```

Ésta es la forma más eficiente de evaluar el valor de una variable `boolean`. Se puede replantear en una forma menos concisa:

```
if (abierta == true) {
```

Para resumir, las variables `boolean` se utilizan en la programación para recordar si algo es verdadero o falso, tal vez por un tiempo corto o tal vez durante todo el tiempo que se ejecute el programa.

## PRÁCTICA DE AUTOEVALUACIÓN

- 7.12** El propietario de la tienda necesita un anuncio adicional que diga “OFERTA”.  
 ¿Podemos seguir utilizando una variable `boolean`?

Los métodos pueden usar valores `boolean` como parámetros y como valores de retorno. Por ejemplo, he aquí un método que verifica si tres números están en orden:

```

private boolean enOrden(int a, int b, int c) {
 if ((a <= b) && (b <= c)) {
 return true;
 }
 else {
 return false;
 }
}

```



## Comparación de cadenas

Hasta ahora hemos visto programas que usan los operadores de comparación (como `>`) para comparar números. Sin embargo, muchos programas necesitan comparar cadenas de texto. En estos casos los operadores de comparación no son apropiados, por lo que en vez de ellos utilizamos el método `equals`.

El programa de la bóveda que vimos antes requería que el usuario introdujera un código numérico. Suponga que ahora el código es alfabético. En este caso, el programa necesita comparar la cadena introducida con el código correcto (por decir, “Bill”). La instrucción `if` apropiada sería:

```
String código;
código = campoCódigo.getText();
if (código.equals("Bill")) {
 campoTextoResultado.setText("abierta");
}
```

Se hace una llamada al método `equals`. El parámetro es la cadena “Bill”. El método devuelve verdadero o falso. Después, la instrucción `if` actúa de manera acorde.

## Fundamentos de programación

Por lo general la computadora lleva a cabo las instrucciones una por una, en secuencia. Una instrucción `if` instruye a la computadora para que evalúe el valor de ciertos datos y después elija una de varias acciones dependiendo del resultado de la evaluación. A este proceso de elección se le conoce algunas veces como selección. A la evaluación de los datos se le conoce como condición. Una vez que se completa una instrucción `if`, la computadora sigue llevando a cabo las instrucciones en secuencia.

## Errores comunes de programación

### Paréntesis

La condición dentro de una instrucción `if` debe ir encerrada entre paréntesis. Por ejemplo:

```
if (a > b) etc.
```

### Igualdad

Si desea evaluar la igualdad, use el operador `==` (y no un solo signo de igual, `=`). El siguiente ejemplo es correcto:

```
if (a == b) etc.
```



*Errores comunes de programación (continuación)*

Por desgracia, un programa que utilice un solo `=` se compilará correctamente pero funcionará en forma incorrecta.

## Comparación de cadenas

Si desea comparar dos cadenas, debe usar el método `equals` como en el siguiente ejemplo:

```
if (cadena1.equals(cadena2)) etc.
```

## Llaves

Ahora veamos las llaves. Esta instrucción es completamente correcta:

```
if (código == 123)
 campoTextoResultado.setText("abierta");
```

aun cuando se omitieron las llaves que rodean a la instrucción. La regla de Java es que si sólo hay **una** instrucción a ejecutar, entonces no son necesarias las llaves. Sin embargo, esto puede ocasionar molestos problemas de programación, por lo que nuestro mejor consejo es insertar las llaves en todo momento. Hay una excepción a esta sugerencia cuando usamos instrucciones `if` anidadas en el estilo `else if`, como explicamos antes.

## Condiciones compuestas

Tal vez alguna vez llegue a escribir una instrucción `if` de esta forma:

```
if (a > 18 && < 25)
```

lo cual es incorrecto. El signo `&&` debe enlazar dos condiciones completas, de preferencia entre paréntesis para mejorar la claridad, como en el siguiente ejemplo:

```
if ((a > 18) && (a < 25))
```

## switch

La instrucción `switch` es muy útil, pero por desgracia no es tan flexible como quisiéramos. Suponga, por ejemplo, que deseamos escribir una pieza de programa para mostrar dos números, con el mayor primero, seguido del menor. Si usamos instrucciones `if` podríamos escribir lo siguiente:

```
if (a > b) {
 campoTexto.setText(Integer.toString(a) + " es mayor que "
 + Integer.toString(b));
}
```

```

if (b > a) {
 campoTexto.setText(Integer.toString(b) + " es mayor que "
 + Integer.toString(a));
}
if (a == b) {
 campoTexto.setText("son iguales");
}

```

Podríamos vernos tentados a rediseñar esto mediante una instrucción **switch**, como se muestra a continuación:

```

switch (?) { // ¡cuidado! Java inválido
 case a > b:
 campoTexto.setText(Integer.toString(a) + " es mayor que "
 + Integer.toString(b));
 break;
 case b > a:
 campoTexto.setText(Integer.toString(b) + " es mayor que "
 + Integer.toString(a));
 break;
 case a == b:
 campoTexto.setText("son iguales");
 break;
}

```

Pero esto no se permite ya que, como lo indica el signo de interrogación, **switch** sólo trabaja con una sola variable entera como sujeto y **case** no puede utilizar los operadores **>**, **==**, **<**, etc.

## Secretos de codificación

El primer tipo de instrucción **if** tiene la siguiente estructura:

```

if (condición) {
 instrucciones
}

```

El segundo tipo de instrucción **if** tiene la siguiente estructura:

```

if (condición) {
 instrucciones
}
else {
 instrucciones
}

```



 *Secretos de codificación (continuación)*

La instrucción **switch** tiene la siguiente estructura:

```
switch (variable) {
 case valor1:
 instrucciones
 break;
 case valor2:
 instrucciones
 break;
 default:
 instrucciones
 break;
}
```

La sección predeterminada (**default**) es opcional.

## Nuevos elementos del lenguaje

- Estructuras de control para decisiones:

```
if, else
switch, case, break, default
```

- Los operadores de comparación **>**, **<**, **==**, **!=**, **<=** y **>=**.
- Los operadores lógicos **&&**, **||** y **!**.
- Las variables declaradas como **boolean**, que pueden contener el valor **true** o el valor **false**.

## Resumen

Las instrucciones **if** permiten al programador controlar la secuencia de acciones al hacer que el programa lleve a cabo una evaluación. Después de realizar la evaluación, la computadora ejecuta una de varias acciones disponibles. Hay dos variedades de una instrucción **if**:

- **if**
- **if...else**

La instrucción **if** se puede utilizar para identificar el origen de un evento de GUI. El método **getSource** devuelve el objeto que produjo el evento. Este objeto se compara con cada uno de los objetos que pudieron haber causado el evento, mediante el operador de comparación **==**.

La instrucción **switch** ofrece una manera conveniente de llevar a cabo varias evaluaciones. Sin embargo, está restringida a realizar evaluaciones con enteros.

Una variable **boolean** puede contener el valor **true** o el valor **false**. Se puede evaluar mediante una instrucción **if**. Una variable **boolean** es útil en situaciones en las que una variable sólo tiene dos valores significativos.

## Ejercicios

- 7.1 Precio de función de cine** Escriba un programa para averiguar cuánto paga una persona por ir a una función de cine. El programa debe recibir una edad desde un control deslizable o un campo de texto y después debe decidir con base en lo siguiente:
- Menor de 5 años, la función es gratis.
  - De 5 a 12, paga la mitad de la tarifa.
  - De 13 a 54, paga la tarifa completa.
  - De 55 o mayor, la función es gratis.
- 7.2 El elevador** Escriba un programa para simular un elevador muy primitivo. Para representar al elevador utilice un cuadrado relleno de color negro que se muestre en un panel alto, delgado y blanco. Debe haber dos botones: uno para que se desplace 20 píxeles hacia arriba por el panel y otro para que baje. Despu  s mejore el programa para asegurarse de que el elevador no suba o baje demasiado.
- 7.3 Ordenamiento** Escriba un programa que reciba n  meros de tres controles deslizables o tres campos de texto y los muestre por tama  o num  rico en orden de menor a mayor.
- 7.4 Apuestas** Un grupo de personas desean apostar sobre el resultado de tres lanzamientos de un dado. Una persona debe apostar \$1 para tratar de adivinar el resultado de los tres lanzamientos. Escriba un programa que utilice el m  todo de los n  meros aleatorios para simular tres lanzamientos de un dado y que muestre las ganancias de acuerdo con las siguientes reglas:
- Los tres lanzamientos cayeron en seis: gana \$20.
  - Los tres lanzamientos cayeron en el mismo n  mero (pero no en seis): gana \$10.
  - Dos de tres lanzamientos cayeron en el mismo n  mero: gana \$5.
- 7.5 B  veda de combinaci  n digital** Escriba un programa que act  e como un candado de combinaci  n digital para una b  veda. Cree tres botones que representen los n  meros 1, 2 y 3. El usuario debe hacer clic en los botones para tratar de adivinar los n  meros correctos (por ejemplo, 331121). El programa debe permanecer sin hacer nada hasta que se opriman los botones correctos. Despu  s debe felicitar al usuario con un mensaje apropiado. Debe haber un bot  n para reiniciar.
- Mejore el programa de manera que tenga otro bot  n para que el usuario pueda modificar la combinaci  n de la b  veda, siempre y cuando se haya introducido el c  digo correcto.
- 7.6 Repartir una carta** Escriba un programa con un solo bot  n que, al ser oprimido, seleccione al azar una carta. Primero utilice el generador de n  meros aleatorios de la biblioteca para crear un n  mero en el rango de 1 a 4. Despu  s convierta el n  mero en un palo (coraz  n, diamante, tr  bol y espada). Luego utilice el generador de n  meros aleatorios para crear un n  mero aleatorio en el rango de 1 a 13. Convierta el n  mero en un as, un 2, 3, etc., y finalmente muestre el valor de la carta elegida.
- Sugerencia: utilice **switch** de la manera m  s apropiada.
- 7.7 Juego de piedra, papel o tijera** En su forma original, cada uno de los dos jugadores elige al mismo tiempo piedra, papel o tijera. La piedra gana a la tijera, el papel gana a la piedra y la tijera gana al papel. Si ambos jugadores eligen la misma opci  n, es un empate. Escriba un programa para jugar este juego. El jugador debe seleccionar uno de tres botones marcados como piedra, papel o tijera. La computadora debe realizar su elecci  n al azar mediante el generador de n  meros aleatorios. Tambi  n debe decidir y mostrar qui  n gana.



**Figura 7.12** La calculadora.

- 7.8 La calculadora** Escriba un programa que simule una calculadora simple de escritorio (figura 7.12), que opere con números enteros. Debe tener un botón para cada uno de los 10 dígitos del 0 al 9. También debe tener un botón para sumar y otro para restar. Además debe contar con un botón **borrar** para borrar la pantalla (un campo de texto) y un botón de igual (=) para obtener la respuesta.

Al oprimir el botón para borrar la pantalla, ésta debe quedar en cero y el total (oculto) debe quedar también en 0.

Al oprimir el botón de un dígito, éste se debe agregar a la derecha de los dígitos que ya se encuentren en la pantalla (en caso de haber alguno).

Al oprimir el botón +, el número en la pantalla se deberá sumar al total (haga lo mismo para el botón -).

Al oprimir el botón = deberá aparecer el valor del total.

- 7.9 Nim** Este juego se juega con cerillos. No importa cuántos cerillos haya. Hay que colocarlos en tres pilas. Tampoco importa cuántos cerillos haya en cada pila. Los jugadores van por turnos. Cada jugador puede quitar cualquier cantidad de cerillos de cualquier pila, pero sólo de una pila. Además, cada jugador debe quitar por lo menos un cerillo. El ganador es la persona que haga que el otro jugador tome el último cerillo.

Escriba un programa para jugar este juego. Al principio la computadora debe repartir tres pilas de cerillos, con un número aleatorio (en el rango de 1 a 200) de cerillos en cada pila. Las tres cantidades se deben mostrar en campos de texto. La computadora debe participar como jugador, debe elegir una pila y una cantidad de cerillos al azar. El otro jugador será el usuario humano, quien debe especificar el número de pila con un botón y la cantidad de cerillos mediante un campo de texto.

También debe haber un botón “nuevo juego”.

- 7.10 Gráficos de tortuga** Este tipo de gráficos es una forma de facilitar a los niños el aprendizaje de la programación. Imagine una pluma fija en la panza de una tortuga. A medida que la tortuga se mueve por el piso, la pluma deja un trazo. La tortuga puede avanzar hacia el norte, sur, este y oeste. Además, la tortuga puede recibir órdenes mediante un botón para cada orden, de la siguiente manera:

- Subir pluma.
- Bajar pluma.
- Voltear a la izquierda 90°.
- Voltear a la derecha 90°.
- Avanzar  $n$  píxeles.

En un principio la tortuga se encuentra en la parte superior izquierda del panel y de cara hacia el este. Con base en esto, podemos dibujar un rectángulo si utilizamos la siguiente secuencia de ejemplo:

- |                                                                                                                                                                         |                                                                                                                                                                                      |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ol style="list-style-type: none"> <li>1. Bajar pluma.</li> <li>2. Avanzar 20 píxeles.</li> <li>3. Voltear a la derecha 90°.</li> <li>4. Avanzar 20 píxeles.</li> </ol> | <ol style="list-style-type: none"> <li>5. Voltear a la derecha 90°.</li> <li>6. Avanzar 20 píxeles.</li> <li>7. Voltear a la derecha 90°.</li> <li>8. Avanzar 20 píxeles.</li> </ol> |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

Escriba un programa que se comporte como la tortuga, con un botón para cada una de las órdenes. Utilice un control deslizable o un cuadro de texto para introducir el número de píxeles ( $n$ ) que debe avanzar la tortuga. La dirección de la tortuga (norte, sur, este, oeste) se debe mostrar en un campo de texto.

## Respuestas a las prácticas de autoevaluación

- 7.1**
- ```
if (código == 123) {
    campoTextoResultado.setText("abierta");
    campoCódigo.setText("");
}
```
- 7.2** No, ya que consideran a la edad específica de 18 de manera distinta.
- 7.3** La parte esencial de este programa es:
- ```
if (temp < min) {
 min = temp;
}
campoTexto.setText("El valor mínimo es " + min);
```
- 7.4**
- ```
public void actionPerformed(ActionEvent event) {
    Object origen;
    origen = event.getSource();
    if (origen == botón1) {
        campoTexto.setText("botón 1");
    }
    if (origen == botón2) {
        campoTexto.setText("botón 2");
    }
    if (origen == botón3) {
        campoTexto.setText("botón 3");
    }
}
```



*Respuestas a las prácticas de autoevaluación (continuación)*

```
7.5    if (edad <= 16) {
        campoTexto.setText("demasiado joven");
    }

7.6    if ((total == 2) || (total == 5) || (total == 7)) {
        campoTexto.setText("usted ha ganado");
    }

7.7    if ((edad >= 16) && (edad < 65)) {
        JOptionPane.showMessageDialog(null, "puede obtener un empleo");
    }

7.8    int salario, impuestos;

        salario = deslizante.getValue();

        if ((salario > 10000) && (salario <= 50000)) {
            impuestos = (salario - 10000) / 5;
        }
        if (salario > 50000) {
            impuestos = 8000 + ((salario - 50000) * 9 / 10);
        }
        if (salario <= 10000) {
            impuestos = 0;
        }

7.9    public void stateChanged(ChangeEvent e) {
        int a, b, c;
        int mayor;

        a = deslizante1.getValue();
        b = deslizante2.getValue();
        c = deslizante3.getValue();

        if ((a >= b) && (a >= c)) {
            mayor = a;
        }
        else if ((b >= a) && (b >= c)) {
            mayor = b;
        }
        else {
            mayor = c;
        }
        JOptionPane.showMessageDialog(null,
            "el valor mayor es " + mayor);
    }
```

```
7.10    int edad;
edad = Integer.parseInt(campoTextoEdad.getText());
if ((edad >= 18) && (edad <= 30)) {
    campoTextoResultado.setText("puede irse de vacaciones");
}

7.11    private String convertir(int p) {
        String palo;

        switch (s) {
            case 1:
                palo = "diamantes";
                break;
            case 2:
                palo = "corazones";
                break;
            case 3:
                palo = "tréboles";
                break;
            case 4:
                palo = "espadas";
                break;
            default:
                palo = "error";
                break;
        }
        return palo;
}
```

7.12 No, porque ahora hay tres posibles valores, no dos.

CAPÍTULO 8



Repetición

En este capítulo conoceremos cómo:

- Realizar repeticiones mediante instrucciones `while`.
- Realizar repeticiones mediante instrucciones `for`.
- Utilizar los operadores lógicos `&&`, `||` y `!` en ciclos.
- Realizar repeticiones mediante la instrucción `do`.

● Introducción

Nosotros los humanos estamos acostumbrados a realizar cosas una y otra vez; por ejemplo: comer, dormir y trabajar. Las computadoras realizan las repeticiones en forma similar. Algunos ejemplos son:

- Sumar una lista de números.
- Buscar cierta información en un archivo.
- Resolver una ecuación matemática en forma iterativa, obteniendo cada vez mejores aproximaciones.
- Hacer que una imagen gráfica se mueva en la pantalla (animación).

Ya hemos visto que una computadora lleva a cabo una secuencia de instrucciones. Ahora veremos cómo repetir una secuencia de instrucciones cierto número de veces. Parte del poder de las computadoras surge de su capacidad para realizar repeticiones con extrema rapidez. En el lenguaje de la programación, a una repetición se le conoce como ciclo.

Hay dos formas principales en las que el programador de Java puede instruir a la computadora para que realice una repetición: `while` y `for`. Podemos utilizar cualquiera de estas dos instrucciones para llevar a cabo repeticiones pero hay diferencias entre ellas, como veremos a continuación.

while

Primero vamos a utilizar un ciclo para mostrar los enteros del 1 al 10 (figura 8.1) en un cuadro de texto, para lo cual usaremos el siguiente programa:



Figura 8.1 Pantalla de los números del 1 al 10.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class UnoAlDiez extends JFrame implements ActionListener {

    private JButton botón;
    private JTextField campoTexto;

    public static void main(String[] args) {
        UnoAlDiez demo = new UnoAlDiez();
        demo.setSize(200, 120);
        demo.crearGUI();
        demo.setVisible(true);
    }

    private void crearGUI() {
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        Container ventana = getContentPane();
        ventana.setLayout(new FlowLayout());

        botón = new JButton("empezar");
        ventana.add(botón);
        botón.addActionListener(this);

        campoTexto = new JTextField(15);
        ventana.add(campoTexto);
    }
}
```

```

public void actionPerformed(ActionEvent event) {
    int número;
    String unoAlDiez = "";
    número = 1;
    while (número <= 10) {
        unoAlDiez = unoAlDiez + Integer.toString(número) + " ";
        número++;
    }
    campoTexto.setText(unoAlDiez);
}

```

La palabra **while** indica que se requiere una repetición. Las instrucciones encerradas entre las llaves se repiten, y a esto se le conoce como el cuerpo del ciclo. La condición entre paréntesis que va inmediatamente después de la palabra **while** controla el ciclo. Si la condición es verdadera, el ciclo continúa. Si es falsa, el ciclo termina y el control se transfiere de vuelta a la instrucción que esté después de la llave de cierre. En este caso el ciclo continúa mientras que cada **número** sea menor o igual que 10.

Antes de iniciar el ciclo, el valor de **número** se hace igual a 1. Al final de cada ciclo, el valor de **número** se incrementa en 1 mediante el operador **++** que vimos en un capítulo anterior. Así, **número** recibe los valores 1, 2, 3, ... hasta el 10.

En un principio la cadena **unoAlDiez** está vacía. Cada vez que se repite el ciclo, se agrega un número (y un espacio) a la cadena mediante el operador de cadena **+**.

El fragmento del programa anterior utiliza el operador menor o igual que (**<=**). Éste es uno de varios operadores de comparación disponibles, los mismos que utilizamos en las instrucciones **if**. He aquí de nuevo la lista completa de los operadores de comparación (a los que se les conoce también como operadores relacionales):

Símbolo	Significado
>	mayor que
<	menor que
==	Igual que
!=	no es igual que
<=	menor o igual que
>=	mayor o igual que

La sangría que se aplica en las instrucciones dentro del ciclo nos ayuda a ver su estructura.

Si su sistema de desarrollo provee un depurador, puede usarlo para seguir con más facilidad la ejecución de este ciclo.

PRÁCTICAS DE AUTOEVALUACIÓN

8.1 ¿Qué hace el siguiente fragmento de programa?

```
String cadena = "";
int número = 0;
while (número <= 5) {
    cadena = cadena + Integer.toString(número * número) + " ";
    número++;
}
areaTexto.setText(cadena);
```

8.2 Escriba un programa que sume (calcule la suma de) los números del 1 al 100 y la muestre en un campo de texto al hacer clic en un botón.



Figura 8.2 Repetición de cajas utilizando `while`.

El siguiente programa utiliza un ciclo `while` para mostrar una fila de cajas (figura 8.2). El número de cajas se determina con base en el valor seleccionado en un control deslizable. Cada vez que se desplaza el deslizador se genera un evento y el programa muestra el número equivalente de cajas. Para hacer esto necesitamos un contador. El contador, que en un principio es igual a 1, se incrementa en 1 cada vez que se muestra una caja. Necesitamos repetir el proceso de mostrar una caja adicional hasta que el contador llegue al total deseado, para lo cual utilizaremos un ciclo `while` como se aprecia a continuación:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;

public class Cajas extends JFrame implements ChangeListener {

    private JSlider deslizante;
    private JPanel panel;
```

```

public static void main(String[] args) {
    Cajas demo = new Cajas();
    demo.setSize(250,150);
    demo.crearGUI();
    demo.setVisible(true);
}

private void crearGUI() {
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    Container ventana = getContentPane();
    ventana.setLayout(new FlowLayout());

    deslizante = new JSlider(0, 10, 1);
    ventana.add(deslizante);
    deslizante.addChangeListener(this);

    panel = new JPanel();
    panel.setPreferredSize(new Dimension(180, 50));
    panel.setBackground(Color.white);
    ventana.add(panel);
}

public void stateChanged(ChangeEvent e) {
    Graphics papel = panel.getGraphics();
    int x, númeroDeCajas, contador;
    númeroDeCajas = deslizante.getValue();
    papel.setColor(Color.white);
    papel.fillRect(0, 0, 180, 50);
    x = 10;
    contador = 1;
    papel.setColor( Color.black);
    while (contador <= númeroDeCajas) {
        papel.drawRect(x, 10, 10, 10);
        x = x + 15;
        contador++;
    }
}
}

```



Este programa dibujará todas las cajas que necesitemos. Imagine cuántas instrucciones tendríamos que escribir para mostrar 100 cajas, si no pudiéramos utilizar una instrucción **while**. Tenga presente también que este programa dibujará cero cajas si es lo que el usuario selecciona con el deslizador. Por ende, la instrucción **while** es completamente flexible: nos proporciona tantas o tan pocas repeticiones como requiramos.

Una forma de visualizar un ciclo **while** es por medio de un diagrama de actividades, como se muestra en la figura 8.3. Por lo general la computadora lleva a cabo las instrucciones en secuencia de arriba hacia abajo, como lo indican las flechas. Un ciclo **while** implica que se debe evaluar la condición antes de ejecutar el ciclo, y se debe evaluar otra vez antes de que el ciclo se repita. Si la condición es verdadera, se ejecuta el ciclo. Cuando por fin la condición es falsa, el cuerpo del ciclo deja de ejecutarse y la repetición termina.

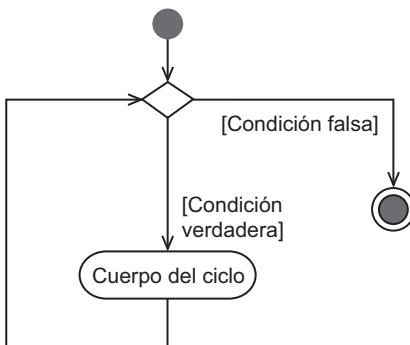


Figura 8.3 Diagrama de actividades del ciclo `while`.

Es conveniente tener mucho cuidado al escribir un ciclo `while` para asegurarnos de que el conteo se realice en forma apropiada. Un error común es hacer que el ciclo se repita demasiadas veces o muy pocas veces. Esto se conoce comúnmente como un error de “desplazamiento por uno” (*off by one*). Algunas veces un ciclo se escribe de manera que empiece con un contador de 0 y la evaluación es para verificar si la condición es menor que el número requerido, como se muestra a continuación:

```

contador = 0;
while (contador < númeroRequerido) {
    // cuerpo
    contador++;
}
  
```

También podemos escribir el ciclo de manera que empiece con un contador de 1 y la evaluación sea para verificar si la condición es menor o igual al número requerido, como se muestra a continuación:

```

contador = 1;
while (contador <= númeroRequerido) {
    // cuerpo
    contador++;
}
  
```

Utilizaremos ambos estilos en este libro.

PRÁCTICAS DE AUTOEVALUACIÓN

- 8.3 **Barras de la prisión** Escriba un programa para dibujar cinco líneas verticales paralelas.
- 8.4 **Tablero de ajedrez** Escriba un programa para dibujar un tablero de ajedrez con nueve líneas verticales con una separación de 10 píxeles, y nueve líneas horizontales con una separación de 10 píxeles.
- 8.5 **Cuadrados de números** Escriba un programa para mostrar los números del 1 al 5 y sus cuadrados, un número (y su cuadrado) por línea en un área de texto. Use la cadena “\n” para avanzar al inicio de una nueva línea.

for

En el ciclo **for**, muchos de los ingredientes de un ciclo **while** se agrupan en el encabezado de la instrucción en sí. Por ejemplo, veamos el programa anterior para mostrar los números del 1 al 10, pero ahora utilizando **for**:

```
unoAlDiez.setText("");
for (int número = 1; número <= 10; número++) {
    unoAlDiez = unoAlDiez + Integer.toString(número) + " ";
}
```

Dentro de los paréntesis de la instrucción **for** hay tres ingredientes separados por signos de punto y coma:

- Una instrucción inicial. Se lleva a cabo sólo una vez, antes de que inicie el ciclo.
Ejemplo: `int número = 1`
- Una condición. Se evalúa antes de cualquier ejecución del ciclo.
Ejemplo: `número <= 10`
- Una instrucción final. Ésta se lleva a cabo al final de cada ciclo, justo antes del final de cada ciclo.
Ejemplo: `número++`

La condición determina si el ciclo **for** se ejecuta o completa de la siguiente manera:

- Si la condición es verdadera, se ejecuta el cuerpo del ciclo.
- Si la condición es falsa, el ciclo termina y se ejecutan las instrucciones que van después de la llave de cierre.

Tenga en cuenta que puede escribir una declaración completa de una variable dentro del encabezado de una instrucción **for** junto con su inicialización, y esto es algo que se hace con frecuencia. Esta variable se puede utilizar a lo largo del cuerpo de la instrucción **for**.

He aquí otro programa de ejemplo que utiliza un ciclo **for** para mostrar 20 círculos pequeños en coordenadas aleatorias en un panel de color negro, como el cielo de la noche (figura 8.4).

```
public void actionPerformed(ActionEvent event) {
    Graphics papel = panel.getGraphics();
    papel.setColor(Color.black);
    papel.fillRect(0, 0, 200, 200);
    papel.setColor(Color.white);
    for (int contador = 0; contador < 20; contador++) {
        int x, y, radio;
        x = aleatorio.nextInt(200);
        y = aleatorio.nextInt(200);
        radio = 5;
        papel.fillOval(x, y, radio, radio);
    }
}
```

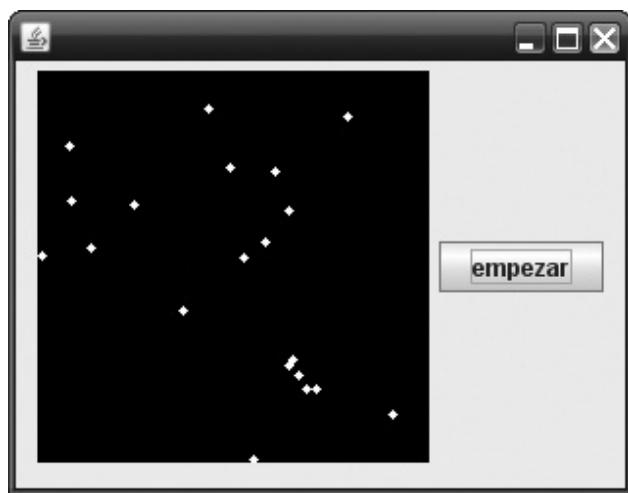


Figura 8.4 El programa de las estrellas.

Siempre es posible volver a codificar un ciclo `for` como un ciclo `while` y viceversa. Pero por lo general uno de los dos será el más claro. El ciclo `for` se utiliza comúnmente cuando contamos, sumando o restando un valor fijo en cada paso.

Además, se considera un estilo pobre terminar el ciclo antes de completar el patrón descrito en el encabezado de un ciclo `for`.

PRÁCTICA DE AUTOEVALUACIÓN

8.6 Vuelva a codificar el programa de las estrellas; esta vez utilice `while` en vez de `for`.

● And, or, not

En ocasiones, la condición que controla un ciclo es más compleja y necesitamos los operadores lógicos `and`, `or` y `not`. Usted podría utilizar estos operadores en su vida diaria si quisiera decir: “Voy a dar un paseo hasta que empiece a llover o sean las 5 de la tarde”. Ya vimos estos operadores en el capítulo 7, en la sección sobre el uso de la instrucción `if` en las decisiones. Estos operadores son:

Símbolo	Significado
<code>&&</code>	and
<code> </code>	or
<code>!</code>	not

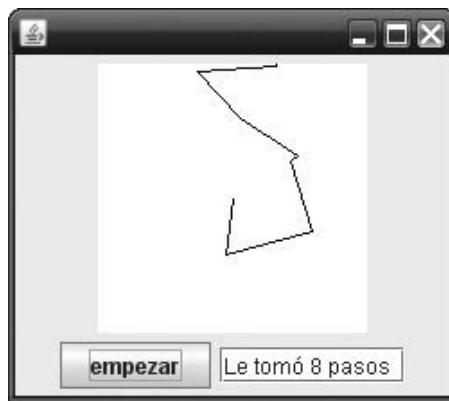


Figura 8.5 Caminata aleatoria.

Si quisieramos describir cuánto tiempo vamos a caminar mediante una instrucción **while**, diríamos: “Mientras que no llueva y no sean las 5 de la tarde, voy a seguir caminando”. Observe que hay un “no” (**not**) antes de cada una de las dos condiciones (llueva, 5 de la tarde) y están enlazadas por un “y” (**and**). Esto es lo que ocurre por lo general cuando usamos un ciclo con una instrucción **while**; es importante tener mucho cuidado en escribir la condición con mucha claridad.

En el siguiente programa utilizamos estos operadores. Una persona ebria intenta llegar a cualquiera de las paredes de un cuarto (representado como un panel). Al principio la persona está en el centro del panel. Da pasos de un tamaño aleatorio y en dirección aleatoria hasta llegar a una pared (figura 8.5).

La posición de la persona en cualquier instante se especifica mediante coordenadas *x* y *y*. El ciclo continúa hasta llegar a uno de los cuatro lados del panel. Por ende, la instrucción **for** implica cuatro condiciones, enlazadas por operadores **&&**. Como verá, ésta es una combinación de condiciones moderadamente compleja. Una combinación como ésta puede ser difícil de escribir y comprender.

```
public void actionPerformed(ActionEvent event) {
    Graphics papel = panel.getGraphics();
    int x, y, pasoX, pasoY, nuevaX, nuevaY, pasos;
    papel.setColor(Color.white);
    papel.fillRect(0, 0, anchuraPapel, alturaPapel);
    x = anchuraPapel / 2;
    y = alturaPapel / 2;
    for (pasos = 0;
        x < anchuraPapel && x > 0
        &&
        y < alturaPapel && y > 0;
        pasos++) {
        pasoX = aleatorio.nextInt(100) - 50;
        pasoY = aleatorio.nextInt(100) - 50;
        nuevaX = x + pasoX;
        nuevaY = y + pasoY;
```

```

    papel.setColor(Color.black);
    papel.drawLine(x, y, nuevaX, nuevaY);

    x = nuevaX;
    y = nuevaY;
}
campoTexto.setText("Le tomó " + pasos + " pasos");
}

```

PRÁCTICA DE AUTOEVALUACIÓN

8.7 ¿Qué aparece en pantalla al ejecutar las siguientes instrucciones?

```

int n, m;
n = 10;
m = 5;
while ((n > 0) || (m > 0)) {
    n = n - 1;
    m = m - 1;
}
 JOptionPane.showMessageDialog(null,
        ("n = " + Integer.toString(n) +
         " m = " + Integer.toString(m)));

```

do...while

Si utiliza las instrucciones **while** o **for**, la evaluación siempre se realiza al principio de la repetición. El ciclo **do** es una estructura alternativa en la cual la evaluación se lleva a cabo al final de cada repetición. Esto significa que el ciclo siempre se repite por lo menos una vez. Ilustraremos el uso del ciclo **do** escribiendo piezas de programas para mostrar los números del **0** al **9** en un campo de texto, utilizando las tres estructuras de ciclo disponibles. El texto se acumula en una cadena de texto.

Si usamos **while**:

```

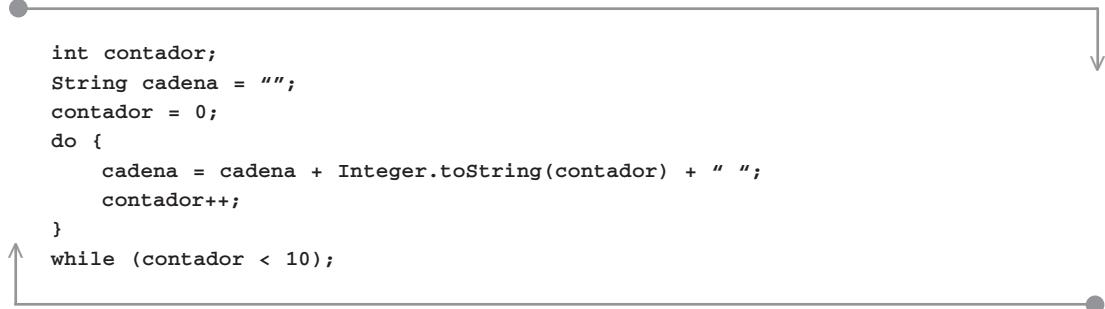
int contador;
String cadena = "";
contador = 0;
while (contador <= 9) {
    cadena = cadena + Integer.toString(contador) + " ";
    contador++;
}

```

Si usamos **for**:

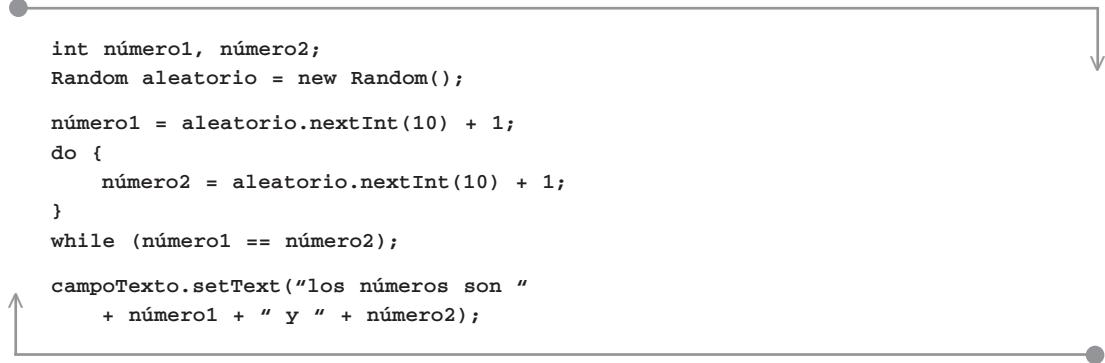
```
String cadena = "";
for (int contador = 0; contador <=9; contador++) {
    cadena = cadena + Integer.toString(contador) + " ";
}
```

Si usamos **do** (la evaluación se hace al final del ciclo):



```
int contador;
String cadena = "";
contador = 0;
do {
    cadena = cadena + Integer.toString(contador) + " ";
    contador++;
}
while (contador < 10);
```

Ahora veremos un ejemplo en donde un ciclo necesita ejecutarse por lo menos una vez. En una lotería los números se seleccionan al azar, pero no puede haber dos números iguales. Consideremos una lotería muy pequeña en la que sólo se seleccionan dos números. Al principio obtenemos el primer número aleatorio. Ahora necesitamos un segundo número, pero éste no debe ser igual al primero. Entonces seleccionamos repetidas veces un segundo número hasta que no sea igual al primero. He aquí el programa:



```
int número1, número2;
Random aleatorio = new Random();

número1 = aleatorio.nextInt(10) + 1;
do {
    número2 = aleatorio.nextInt(10) + 1;
}
while (número1 == número2);

campoTexto.setText("los números son "
+ número1 + " y " + número2);
```

Ciclos anidados

Un ciclo anidado es un ciclo dentro de otro ciclo. Suponga, por ejemplo, que deseamos mostrar la pantalla de la figura 8.6, que es un bloque de apartamentos dibujados en forma muy rudimentaria. El tamaño del edificio se determina con base en la posición de los controles deslizables. Suponga que hay cuatro pisos, cada uno con cinco apartamentos, los cuales aparecen como rectángulos. El ciclo para dibujar un piso individual tiene la siguiente estructura:

```
for (int apartamento = 1; apartamento <= 5; apartamento++) {  
    // código para dibujar un apartamento  
}
```

y el ciclo para dibujar varios pisos tiene la siguiente estructura:

```
for (int piso = 1; piso <= 3; piso++) {  
    // código para dibujar un piso  
}
```

Lo que necesitamos es encerrar el primer ciclo dentro del segundo, de manera que los ciclos estén anidados. Podemos usar controles deslizables para establecer el número de apartamentos por piso y el número de pisos. Cada vez que modifiquemos cualquiera de los controles deslizables se producirá un evento y se ejecutarán las siguientes instrucciones:

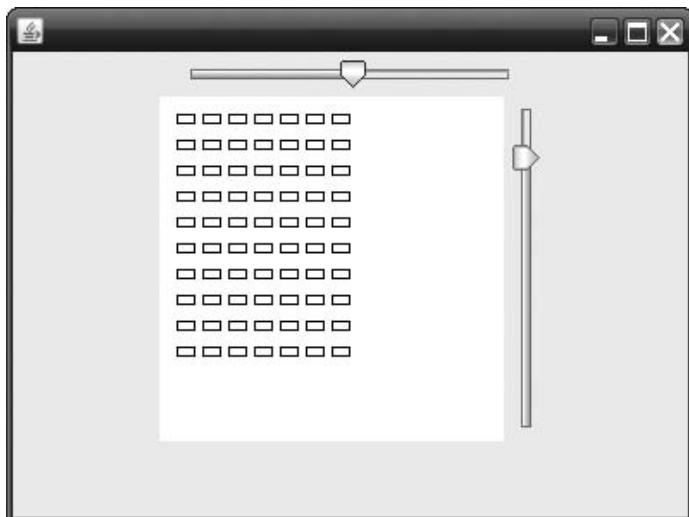


Figura 8.6 Pantalla del bloque de apartamentos.

```

int pisos, apartamentos;
int x, y;
y = 10;
papel.setColor(Color.white);
papel.fillRect(0, 0, 200, 200);

apartamentos = deslizante1.getValue();
pisos = deslizante2.getValue();
for (int piso = 0; piso <= pisos; piso++) {
    x = 10;
    for (int contador = 0; contador <= apartamentos; contador++) {
        papel.setColor(Color.black);
        papel.drawRect(x, y, 10, 5);
        x = x + 15;
    }
    y = y + 15;
}

```

En el ejemplo anterior podemos ver que la sangría ayuda de manera considerable a comprender el funcionamiento del programa. Siempre es posible rediseñar los ciclos anidados utilizando métodos, que algunas veces son más claros. En el capítulo 19 exploraremos esto con más detalle, cuando hablemos sobre el estilo.

PRÁCTICA DE AUTOEVALUACIÓN

- 8.8** Una partitura de música se escribe en papel impreso con pentagramas. Cada pentagrama consiste en 5 líneas horizontales a lo largo de la página, con espacios de aproximadamente 2 mm (1/10 de pulgada). Cada página contiene ocho de estos pentagramas. Escriba un programa para dibujar una página de partitura musical.

Cómo combinar las estructuras de control

En el capítulo anterior vimos cómo implementar la selección mediante las instrucciones `if` y `switch`; en este capítulo vimos la repetición utilizando `while`, `for` y `do`. La mayoría de los programas consisten en combinaciones de estas estructuras de control. De hecho, la mayoría de los programas constan de:

- Secuencias.
- Ciclos.
- Selecciones.
- Llamadas a métodos de biblioteca.
- Llamadas a métodos que nosotros, los programadores, escribimos.

Fundamentos de programación

Las repeticiones se utilizan mucho en la programación. Hay tres variedades de instrucciones de ciclos en Java: **while**, **for** y **do**. Entonces, ¿cuál debemos usar?

Las instrucciones **while** y **for** son similares, pero **for** se utiliza generalmente cuando hay un contador asociado con el ciclo. Por ende, el prototipo del ciclo **for** tiene la siguiente estructura:

```
for (int contador = 0; contador <= valorFinal; contador++) {  
    // cuerpo del ciclo  
}
```

El contador tiene un valor inicial, un valor final y se incrementa cada vez que se repite el ciclo.

El ciclo **while** se utiliza cuando no se puede calcular el número de repeticiones por adelantado; el ciclo continúa mientras que cierta condición sea verdadera. Un ejemplo es el programa de caminata aleatoria que vimos antes, en donde la repetición continúa hasta que la persona llega a la pared. El prototipo del ciclo **while** tiene la siguiente estructura:

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

El ciclo **do** se utiliza cuando la evaluación para que termine la repetición se necesita hacer al final del ciclo; es importante recordar que un ciclo **do** siempre se lleva a cabo por lo menos una vez.

Los ciclos demuestran su valor en los programas que procesan colecciones de datos. Más adelante en el libro veremos varias colecciones, incluyendo listas, cadenas de texto, archivos y arreglos. También conoceremos una variedad del ciclo **for** (conocida como ciclo **for mejorado**) que es especialmente útil para procesar colecciones.

Errores comunes de programación

Siempre es necesario tener mucho cuidado al escribir la condición en un ciclo. Un error muy común es hacer que un ciclo termine una repetición antes o que se repita demasiadas veces. A esto se le conoce algunas veces como error de “desplazamiento por uno” (*off by one*).

Tenga cuidado con las condiciones complejas en los ciclos. Por ejemplo, ¿necesita **||** o **&&**?

Si el cuerpo de un ciclo es una sola instrucción, no necesita estar rodeado de llaves. Pero por lo general es más seguro utilizarlas, y ésta es la metodología que seguimos en este libro.

Secretos de codificación

El ciclo **while** tiene la siguiente estructura:

```
while (condición) {  
    instrucción(es)  
}
```

en donde la condición se evalúa antes de cualquier repetición del ciclo. Si es verdadera, el ciclo continúa. Si es falsa, el ciclo termina.

El ciclo **for** tiene la siguiente estructura:

```
for (acción inicial; condición; acción) {  
    instrucción(es)  
}
```

en donde:

- **acción inicial** se lleva a cabo una vez, antes de ejecutar el ciclo.
- **condición** se evalúa antes de cada repetición. Si es verdadera, se repite el ciclo; si es falsa, el ciclo termina.
- **acción** se lleva a cabo al final de cada repetición.

El ciclo **do** tiene la siguiente estructura:

```
do {  
    instrucción(es)  
}  
while (condición)
```

La evaluación se realiza después de cada repetición.

Nuevos elementos del lenguaje

Las estructuras de control para repetición:

- **while**
- **for**
- **do**

Resumen

- En programación, a una repetición se le conoce como ciclo.
- Hay tres formas en Java para indicar a la computadora que realice un ciclo: `while`, `for` y `do`.
- La instrucción `for` se utiliza cuando queremos describir las características principales del ciclo dentro de la misma instrucción del ciclo.
- La instrucción `do` se utiliza cuando se debe evaluar una condición al final de un ciclo y/o cuando el ciclo se debe realizar por lo menos una vez.

Ejercicios

- 8.1 Cubos** Escriba un programa que utilice un ciclo para mostrar los números enteros del 1 al 10 junto con los cubos de cada uno de sus valores.
- 8.2 Números aleatorios** Escriba un programa para mostrar 10 números aleatorios mediante un ciclo. Use la clase de biblioteca `Random` para obtener números aleatorios enteros en el rango de 0 a 9. Muestre los números en un campo de texto.
- 8.3 La vía láctea** Escriba un programa que dibuje 100 círculos en un panel en posiciones aleatorias y con diámetros aleatorios de hasta 10 píxeles.
- 8.4 Escalones** Escriba un programa para dibujar una serie de escalones hechos a partir de ladrillos, como se muestra en la figura 8.7. Use el método de biblioteca `drawRect` para dibujar cada ladrillo.



Figura 8.7 Escalones.

- 8.5 Suma de los enteros** Escriba un programa que sume los números del 0 al 39 utilizando un ciclo. Compruebe que ha obtenido la respuesta correcta, utilizando la fórmula para la suma de los números de 0 a n :

$$\text{suma} = n \times (n + 1)/2$$

- 8.6 Patrón de diente de sierra** Escriba un programa para mostrar un patrón de diente de sierra en un cuadro de texto, como se muestra en la figura 8.8. El programa debe utilizar la cadena “\n” para obtener una nueva línea.

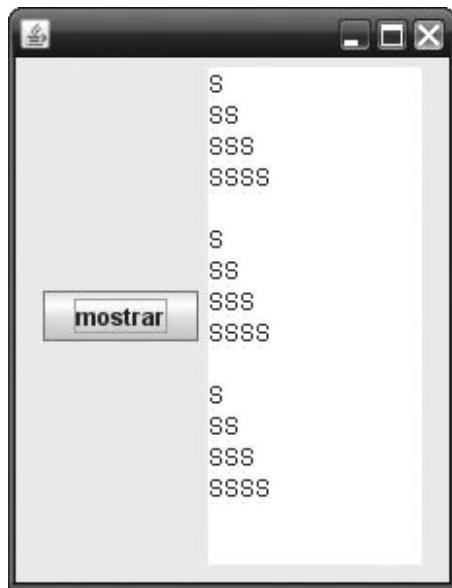


Figura 8.8 Patrón de diente de sierra.

- 8.7 Tabla de multiplicación** Escriba un programa para mostrar una tabla de multiplicación, como la que utilizan los niños pequeños. Por ejemplo, la tabla para los números hasta el 4 se muestra en la figura 8.9. Además de utilizar la cadena “\n” para obtener una nueva línea, el programa debe usar la cadena “\t” para avanzar el tabulador a la siguiente posición, de manera que la información se muestre en columnas ordenadas.

	1	2	3	4
1	1	2	3	4
2	2	4	6	8
3	3	6	9	12
4	4	8	12	16

Figura 8.9 Tabla de multiplicación.

El programa debe ser capaz de mostrar una tabla de cualquier tamaño, el cual se debe especificar introduciendo un entero en un campo de texto. Use el método `setTabSize` para controlar la distribución. Una llamada típica sería:

```
areaTexto.setTabSize(4);
```

- 8.8 Suma de series** Escriba un programa para calcular y mostrar la suma de las series:

$$1 - 1/2 + 1/3 - 1/4 + \dots$$

hasta llegar a un término que sea menor a 0.0001 (la respuesta es aproximadamente 0.6936).

- 8.9 Canción de cuna** Escriba un programa para mostrar todos los versos de una canción de cuna en un área de texto con una barra de desplazamiento vertical. El primer verso es:

10 botellas verdes, colgando en la pared,
10 botellas verdes, colgando en la pared,
Si 1 botella verde se fuera a caer,
Habría 9 botellas verdes, colgando en la pared.

En los versos subsiguientes hay cada vez menos botellas, a medida que se caen de la pared.

Respuestas a las prácticas de autoevaluación

- 8.1** Muestra los números 0 1 4 9 16 25.

- 8.2** La parte central del programa es:

```
int número;
int suma;

suma = 0;
número = 1;
while (número <= 100) {
    suma = suma + número;
    número++;
}
campoTexto.setText("La suma es " + Integer.toString(suma));
```

Este programa utiliza una técnica de programación común: un total acumulado. Al principio el valor de la variable `suma` es igual a cero. Cada vez que se repite el ciclo, el valor de `número` se suma al valor de `suma` y el resultado se coloca nuevamente en `suma`.

- 8.3**
- ```
int x, contador, númeroDeBarras;
númeroDeBarras = 5;
x = 10;
contador = 1;
while (contador <= númeroDeBarras) {
 papel.drawLine(x, 10, x, 100);
 x = x + 15;
 contador++;
}
```





*Respuestas a las prácticas de autoevaluación (continúa)*

**8.4**

```
int x, y, contador;
x = 10;
contador = 1;
while (contador <= 9) {
 papel.drawLine(x, 10, x, 90);
 x = x + 10;
 contador++;
}

y = 10;
contador = 1;
while (contador <= 9) {
 papel.drawLine(10, y, 90, y);
 y = y + 10;
 contador++;
}
```

**8.5**

```
int número = 1;
while (número <= 5) {
 areaTexto.append(Integer.toString(número) + " "
 + Integer.toString(número * número) + "\n");
 número++;
}
```

**8.6** La esencia del ciclo es:

```
int contador = 0;
while (contador < 20) {
 // cuerpo del ciclo
 contador++;
}
```

**8.7**  $n = 0$  y  $m = 5$ .

**8.8**

```
int y;
papel.setColor(Color.white);
papel.fillRect(0, 0, 150, 100);

y = 10;
for (int pentagramas = 1; pentagramas <= 8; pentagramas++) {
 for (int líneas = 1; líneas <= 5; líneas++) {
 papel.setColor(Color.black);
 papel.drawLine(10, y, 90, y);
 y = y + 2;
 }
 y = y + 5;
}
```

# CAPÍTULO 9



## Cómo escribir clases

En este capítulo conoceremos cómo:

- Escribir una clase.
- Escribir métodos **public**.
- Utilizar variables **private** dentro de una clase.
- Escribir métodos constructores.

### ● Introducción

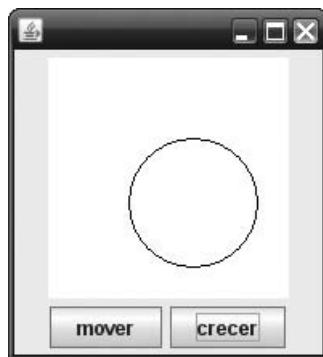
En capítulos anteriores vimos cómo utilizar las clases de biblioteca. En este capítulo veremos cómo escribir nuestras propias clases. Una clase describe cualquier cantidad de objetos similares que se pueden fabricar a partir de ella, mediante la palabra clave **new**.

En este capítulo veremos que una clase consiste por lo general en:

- Datos **private** (variables) que contienen información sobre el objeto.
- Métodos **public** que el usuario del objeto puede llamar para llevar a cabo funciones útiles.
- Opcionalmente uno o más métodos constructores, que se utilizan al momento de crear un objeto. Estos objetos se usan para llevar a cabo cualquier inicialización, como asignar valores iniciales a las variables dentro del objeto.
- Métodos **private** que se utilizan sólo dentro del objeto y son inaccesibles desde su exterior.

## Cómo diseñar una clase

Cuando el programador piensa en un nuevo programa, tal vez surja la necesidad de tener un objeto que no esté disponible en la biblioteca de clases de Java. Primero vamos a utilizar un programa para mostrar y manipular un globo simplificado, y representaremos este globo como un objeto. El programa simplemente muestra un globo como un círculo dentro de un panel, como se muestra en la figura 9.1. Hay dos botones para cambiar la posición y el tamaño del globo.

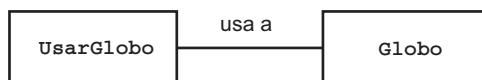


**Figura 9.1** El programa del globo.

Construiremos este programa a partir de dos objetos y, por ende, de dos clases:

- La clase **Globo** representa el globo. Esta clase proveerá los métodos llamados **moverDerecha** y **cambiarTamaño**, cuyas funciones son obvias.
- La clase **UsarGlobo** provee la GUI para el programa. Esta clase utilizará la clase **Globo** según sea necesario.

Ambas clases se muestran en el diagrama de clases de UML de la figura 9.2. Cada una de las clases se representa mediante un rectángulo. Una relación entre las clases se muestra como una línea que une ambas clases. En este caso la relación se muestra como una anotación arriba de la línea: la clase **UsarGlobo** usa a la clase **Globo**.



**Figura 9.2** Diagrama de clases que muestra las dos clases en el programa del globo.

Primero completaremos la clase **UsarGlobo** y después escribiremos la clase **Globo**. El código completo para **UsarGlobo** es:

```

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

```

```
public class UsarGlobo extends JFrame
 implements ActionListener {

 private JButton botónCrecer, botónMover;
 private JPanel panel;

 private Globo globo;

 public static void main(String[] args) {
 UsarGlobo demo = new UsarGlobo();
 demo.setSize(200, 220);
 demo.crearGUI();
 demo.setVisible(true);
 }

 private void crearGUI() {
 setDefaultCloseOperation(EXIT_ON_CLOSE);
 Container ventana = getContentPane();
 ventana.setLayout(new FlowLayout());

 panel = new JPanel();
 panel.setPreferredSize(new Dimension(150, 150));
 panel.setBackground(Color.white);
 ventana.add(panel);

 botónMover = new JButton("mover");
 ventana.add(botónMover);
 botónMover.addActionListener(this);

 botónCrecer = new JButton("crecer");
 ventana.add(botónCrecer);
 botónCrecer.addActionListener(this);

 globo = new Globo();
 }

 public void actionPerformed(ActionEvent event) {
 Graphics papel = panel.getGraphics();
 if (event.getSource() == botónMover) {
 globo.moverDerecha(20);
 }
 else {
 globo.cambiarTamaño(20);
 }
 papel.setColor(Color.white);
 papel.fillRect(0, 0, 150, 150);
 globo.mostrar(papel);
 }
}
```

En la parte superior de la clase **UsarGlobo** declaramos las variables de instancia en la forma usual, incluyendo una variable llamada **globo**:

```
private Globo globo;
```

Dentro de la clase **UsarGlobo** realizamos la inicialización necesaria, incluyendo la creación de una nueva instancia de la clase **Globo**. Éste es el punto crucial en el que creamos un objeto a partir de nuestra propia clase.

```
globo = new Globo();
```

Ahora viene el código para responder a los eventos de clic de botón. Si se hace clic en el botón **mover**, entonces se hace una llamada al método **moverDerecha**. Si se hace clic en el otro botón se hace una llamada a **cambiarTamaño**.

```
public void actionPerformed(ActionEvent event) {
 Graphics papel = panel.getGraphics();
 if (event.getSource() == botónMover) {
 globo.moverDerecha(20);
 }
 else {
 globo.cambiarTamaño(20);
 }
 papel.setColor(Color.white);
 papel.fillRect(0, 0, 150, 150);
 globo.mostrar(papel);
}
```

Éste es todo el código de la clase **UsarGlobo**. Escribir este código nos ayuda a entender la forma en que se utiliza un objeto globo, al tiempo que nos permite ver qué métodos necesita proveer la clase **Globo**, así como la naturaleza de sus parámetros. Esto nos conduce a escribir el código para la clase **Globo**:

```
public class Globo {
 private int x = 50;
 private int y = 50;
 private int diámetro = 20;

 public void moverDerecha(int pasox) {
 x = x + pasox;
 }

 public void cambiarTamaño(int cambio) {
 diámetro = diámetro + cambio;
 }

 public void mostrar(Graphics papel) {
 papel.setColor(Color.black);
 papel.drawOval(x, y, diámetro, diámetro);
 }
}
```

El encabezado de una clase empieza con la palabra clave `class` y proporciona el nombre de la clase, seguido de una llave. La clase completa termina con una llave. Una clase se etiqueta como `public` con la finalidad de que se pueda utilizar ampliamente. La convención de Java (y de la mayoría de los lenguajes OO) es que los nombres de las clases empiecen con mayúscula. El cuerpo de una clase consiste en las declaraciones de las variables y los métodos. Observe cómo se mejora la legibilidad de la clase utilizando sangría y líneas en blanco. En las siguientes secciones del capítulo veremos con detalle cada uno de los ingredientes de la clase anterior.

En resumen, la estructura general de una clase es:

```
public class Globo {
 // variables de instancia
 // métodos
}
```

Ahora que hemos escrito la clase `Globo`, podemos crear cualquier número de instancias de ella. Ya hemos creado un objeto mediante el siguiente código:

```
globo = new Globo();
```

pero además podemos hacer lo siguiente:

```
Globo globo2 = new Globo();
```

## Clases y archivos

Cuando un programa consta sólo de una clase, ya hemos visto que el código fuente de Java se debe colocar en un archivo que tenga el mismo nombre de la clase, pero con la extensión `.java`. Por ejemplo, una clase llamada `Juego` va en un archivo llamado `Juego.java` y el encabezado de la clase es:

```
public class Juego etc.
```

Las instrucciones `import` deben ir antes de este encabezado. El compilador traduce el código de Java en código de bytes y después lo coloca en un archivo llamado `Juego.class`.

Cuando un programa consta de dos o más clases, hay dos metodologías distintas para colocar las clases en archivos:

1. Colocar todas las clases en un solo archivo.
2. Colocar cada clase en su propio archivo.

Ahora bien, los detalles sobre el uso de cada una de estas metodologías dependerán del sistema de desarrollo que usted utilice. Pero he aquí algunos escenarios comunes.

### Un solo archivo

Para adoptar esta metodología:

1. Coloque todas las clases en un archivo.
2. Declare como `public` la clase que contiene el método `main`.

3. Declare todas las demás clases de manera que no sean **public**; es decir, sin descripción de acceso.
4. Haga el nombre del archivo igual al nombre de la clase **public**.
5. Coloque las instrucciones **import** al principio del archivo. Éstas aplican para todas las clases en el archivo.

Por ejemplo, el archivo **UsarGlobo.java** contiene ambas clases:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class UsarGlobo extends JFrame
 implements ActionListener {
 // cuerpo de la clase UsarGlobo
}

class Globo {
 // cuerpo de la clase Globo
}
```

Esta metodología tiene la ventaja de que todas las clases están en un solo lugar. Además, las instrucciones **import** sólo se necesitan una vez.

## Archivos separados

Para adoptar esta metodología:

1. Coloque cada clase en un archivo por sí sola. La mayoría de los entornos de desarrollo interactivos tienen una nueva herramienta de clases.
2. Declare todas las clases como **public**.
3. Haga cada nombre de archivo igual al nombre de la clase que contiene.
4. Coloque las instrucciones **import** apropiadas al inicio de cada clase.
5. Coloque todos los archivos en la misma carpeta.

Por ejemplo, el archivo **UsarGlobo.java** contiene:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class UsarGlobo extends JFrame
 implements ActionListener {

 // cuerpo de la clase UsarGlobo
}
```

Y un segundo archivo llamado **Globo.java** contiene:

```
import java.awt.*;
public class Globo {
 // cuerpo de la clase Globo
}
```

Esta metodología tiene la ventaja de que las clases están en distintos archivos y, por lo tanto, pueden reutilizarse con más facilidad. Utilizamos esta metodología en todo el libro.

Es imprescindible compilar los archivos en orden de dependencia. La clase **UsarGlobo** utiliza un objeto **Globo** y por lo tanto debemos compilar primero la clase **Globo**. Un IDE hará esto de manera automática.

## ● Variables **private**

Un globo tiene ciertos datos asociados: su tamaño (diámetro) y su posición (como coordenadas *x*, *y*). Un objeto globo debe recordar estos valores. Estos datos se guardan en variables que se describen de la siguiente forma:

```
private int x = 50;
private int y = 50;
private int diámetro = 20;
```

Las variables **diámetro**, **x** y **y** se declaran en la parte superior de la clase. Cualquier instrucción dentro de la clase puede acceder a ellas. Se denominan *variables a nivel de clase* o *variables de instancia*.

Hemos agregado la palabra clave **private** a la palabra que normalmente se utiliza para introducir las variables (como **int**). Las variables a nivel de clase casi siempre se declaran como **private**. Esto significa que son accesibles desde cualquier parte dentro de la clase, pero son inaccesibles desde el exterior.

Aunque **podríamos** describir estas variables como **public**, por lo general se considera una mala práctica. Es mejor dejarlas como **private** y usar métodos para acceder a sus valores desde el exterior de la clase. Pronto veremos cómo hacerlo.

## PRÁCTICA DE AUTOEVALUACIÓN

- 9.1 Extienda el objeto globo de manera que tenga una variable que describa el color del globo.

## ● Métodos **public**

Ciertas características de un objeto necesitan estar públicamente disponibles para otras piezas del programa. Esto incluye aquellos métodos que, después de todo, han sido diseñados para que otros métodos los utilicen. Como hemos visto, un globo tiene asociadas ciertas acciones; por ejemplo, para cambiar su tamaño. Estas acciones se escriben como métodos. Para cambiar el tamaño utilizamos el siguiente código:

```
public void cambiarTamaño(int cambio) {
 diámetro = diámetro + cambio;
}
```

Para indicar que el método está públicamente disponible para los usuarios de la clase, debemos anteponer al encabezado del método la palabra clave **public** de Java.

Ahora vamos a escribir el método para mover un globo:

```
public void moverDerecha(int pasoX) {
 x = x + pasoX;
}
```

Para completar la clase proveeremos un método para que un globo se muestre a sí mismo cada vez que se lo soliciten:

```
public void mostrar(Graphics papel) {
 papel.setColor(Color.black);
 papel.drawOval(x, y, diámetro, diámetro);
}
```

Ahora hemos marcado con toda claridad la diferencia entre los elementos públicamente disponibles y los que son privados. Éste es un importante ingrediente de la filosofía de la POO. Los datos (variables) y las acciones (métodos) están agrupados de manera conjunta, pero de tal forma que se oculte parte de la información al mundo exterior. Por lo general, son los datos los que se ocultan al resto del mundo. Esto se denomina *encapsulamiento* u *ocultamiento de información*.

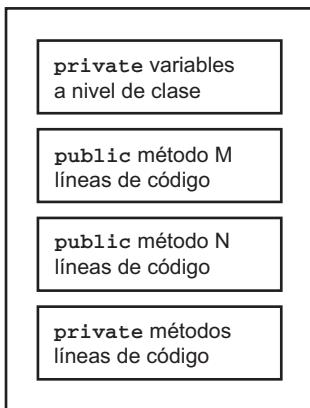
Así, una clase generalmente consta de:

- Métodos **public**.
- Variables **private**.

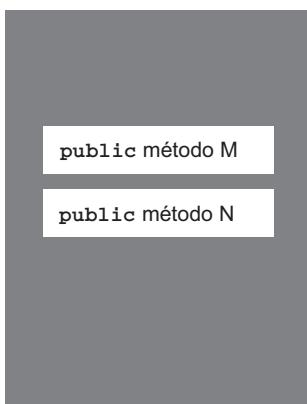
## PRÁCTICAS DE AUTOEVALUACIÓN

- 9.2** Escriba un método que mueva un globo hacia arriba cierta cantidad de espacio que se proporcionará como el parámetro. Asigne a este método el nombre **moverArriba**.
- 9.3** Escriba un método que permita cambiar el color de un globo.
- 9.4** Reescriba el método **mostrar** de manera que muestre un globo coloreado.

Una clase (u objeto) tiene la estructura general que se muestra en la figura 9.3. Ésta es la vista del programador que la escribió; consta de variables y métodos. La forma en que los usuarios ven a un objeto se muestra en la figura 9.4. Los usuarios, a los cuales provee un servicio, ven al objeto en forma muy distinta. Únicamente los elementos **public** (por lo general los métodos) son visibles; todo lo demás está oculto dentro de una caja impenetrable.



**Figura 9.3** Estructura de un objeto o clase en la forma en que lo ve el programador que lo escribió.



**Figura 9.4** Estructura de un objeto o clase en la forma en que lo ven sus usuarios.

## ● Los métodos get y set

Es muy mala práctica permitir el acceso externo a las variables dentro de un objeto. Por ejemplo, suponga que una clase necesita conocer la coordenada **x** de un globo. Sería muy tentador declarar simplemente el valor **x** como **public**. Así el usuario del objeto sólo tendría que hacer referencia al valor como **globo.x**. Esto es factible pero es un mal diseño. Lo que debemos hacer es controlar el acceso a las variables ofreciendo métodos que tengan acceso a ellas. Con esta finalidad, proveemos un método llamado **getX** como parte de la clase **Globo**. Un usuario podría utilizarlo de la siguiente forma:

```
int posición = globo.getX();
```

He aquí el código para el método `getX`:

```
public int getX() {
 return x;
}
```

En general, los usuarios necesitan leer el valor de una variable o modificarlo, o ambas cosas. Por lo tanto, requerimos un método para suministrar el valor, que por convención se denomina método *get*; además necesitamos un método para modificar ese valor, que por convención se denomina método *set*. Las palabras “*get*” y “*set*” no son palabras clave de Java.

Los métodos `getText` y `setText` de la clase `JTextField` son ejemplos comunes de métodos *get* y *set* que se utilizan con mucha frecuencia y que pertenecen a las bibliotecas de Java.

Hay varias razones por las cuales es preferible utilizar los métodos para controlar el acceso a las variables:

- La clase puede ocultar la representación interna de los datos a los usuarios, sin dejar de mantener la interfaz externa. Por ejemplo, el autor de la clase `Globo` podría optar por guardar las coordenadas del punto central de un globo, pero proveer a los usuarios las coordenadas de la esquina superior izquierda de un cuadrado circundante.
- El autor de la clase puede optar por restringir el acceso a los datos. Por ejemplo, la clase podría restringir el valor de la coordenada *x* para que tuviera acceso de sólo lectura (*get*), deshabilitando el acceso de escritura (*set*).
- La clase puede validar o comprobar los valores utilizados. Por ejemplo, podría ignorar un intento de proporcionar un valor negativo para una coordenada.

## PRÁCTICA DE AUTOEVALUACIÓN

- 9.5** Escriba un método para permitir que un usuario sólo tenga acceso *get* a la coordenada *y* de un globo.

## Constructores

Al crear un objeto globo, su posición y tamaño deben recibir valores. A esto se le conoce como inicializar las variables. Hay dos formas de inicializar variables. Una de ellas es incluir la inicialización como parte de la declaración de las variables a nivel de clase. Por ejemplo:

```
private int x = 50;
private int y = 50;
private int diámetro = 20;
```

Otra manera de inicializar un objeto es escribir un método especial para realizar la inicialización. A éste se le conoce como *método constructor* o simplemente *constructor* (ya que está involucrado en la construcción del objeto). Este método siempre debe tener el mismo nombre que la clase. No tiene

valor de retorno, pero por lo general tendrá parámetros. He aquí un método constructor para la clase **Globo**:

```
public Globo(int xInicial, int yInicial,
 int diámetroInicial) {
 x = xInicial;
 y = yInicial;
 diámetro = diámetroInicial;
}
```

Este método asigna los valores de los parámetros (el tamaño y la posición) a las variables apropiadas dentro del objeto. Un método constructor tal como éste se escribe en la parte superior de la clase, después de las declaraciones de las variables a nivel de clase. Los constructores se etiquetan como **public** debido a que se debe acceder a ellos desde el exterior de la clase. Observe que el constructor no tiene un valor de retorno, indicado por la palabra clave **void**.

El método constructor anterior se utilizaría en forma similar al siguiente ejemplo:

```
Globo globo = new Globo(10, 10, 50);
```

Si el programador no inicializa una variable en forma explícita, el sistema de Java le proporciona a cada una de éstas un valor predeterminado. Para los números es cero, para las variables **boolean** es **false**, para las variables **String** es **""** (una cadena vacía) y para cualquier objeto es **null**. Sin embargo, se considera una mala práctica depender de este método de inicialización de variables. Es mejor hacerlo en forma explícita, ya sea al momento de declarar la información o mediante instrucciones dentro de un constructor.

Otras de las acciones que podría realizar un método constructor incluyen la creación de otros objetos que el objeto en cuestión utilice, o abrir un archivo que el objeto utilice.

Si una clase no tiene un constructor explícito, entonces se asume que tiene un solo constructor con cero parámetros, al cual se le conoce como constructor predeterminado o constructor con cero argumentos.

## Varios constructores

Una clase puede tener ninguno, uno o varios métodos constructores. Si tiene uno o más constructores, por lo general llevan parámetros y se deben llamar con los parámetros apropiados. Por ejemplo, en la clase **Globo** podemos escribir los siguientes dos constructores:

```
public Globo(int xInicial, int yInicial,
 int diámetroInicial) {
 x = xInicial;
 y = yInicial;
 diámetro = diámetroInicial;
}
```

```

public Globo(int xInicial, int yInicial) {
 x = xInicial;
 y = yInicial;
 diámetro = 20;
}

```

Lo cual nos permitiría crear objetos globo en cualquiera de las dos formas siguientes:

```

Globo globo1 = new Globo(10, 10, 50);
Globo globo2 = new Globo(10, 10);

```

pero no podríamos hacer esto:

```
Globo globo3 = new Globo();
```

Por lo tanto, si escribe varios constructores pero de todas formas necesita un constructor sin parámetros, debe escribirlo en forma explícita, por ejemplo:

```

public Globo() {
 x = 50;
 y = 50;
 diámetro = 20;
}

```

Ahora tenemos tres constructores para la clase `Globo` y he aquí cómo podríamos usarlos para crear tres objetos distintos a partir de la misma clase:

```

Globo globo1 = new Globo(10, 10, 50);
Globo globo2 = new Globo(10, 10);
Globo globo3 = new Globo();

```

## PRÁCTICA DE AUTOEVALUACIÓN

**9.6** Escriba un método constructor para crear un nuevo globo, especificando sólo el diámetro.

### Métodos `private`

El fin de escribir una clase es permitir la creación de objetos que presenten herramientas útiles a otros objetos. Estas herramientas son los métodos `public` que ofrece el objeto. Pero a menudo una clase tiene métodos que no necesitan hacerse `public` y, de hecho, todos los métodos de los programas que hemos visto en el libro hasta este momento son `private`.

He aquí una clase `Pelota` que representa una pelota que se puede animar de manera que rebote alrededor de un panel. Utiliza métodos `private` así como un método `public` y un constructor. Usa los métodos `private` como forma de aclarar lo que de otra manera sería una pieza compleja de programa. El método `public animar` se llama a intervalos regulares frecuentes para volver a dibujar una imagen. Éste a su vez llama a los métodos `private mover`, `rebotar`, `eliminar` y

**dibujar**. Hemos creado métodos **private** con la finalidad de apoyar a los métodos **public** de la clase. En este ejemplo, los métodos **private** no utilizan parámetros pero, en general, los métodos **private** tienen parámetros.

```
import java.awt.*;
import javax.swing.*;
public class Pelota {

 private JPanel panel;
 private int x = 7, cambioX = 7;
 private int y = 0, cambioY = 2;
 private final int diámetro = 10;
 private final int anchura = 100, altura = 100;

 public Pelota(JPanel elPanel) {
 panel = elPanel;
 }

 public void animar() {
 eliminar();
 mover();
 rebotar();
 dibujar();
 }

 private void mover() {
 x = x + cambioX;
 y = y + cambioY;
 }

 private void rebotar() {
 if (x <= 0 || x >= anchura)
 cambioX = -cambioX;

 if (y <= 0 || y >= altura)
 cambioY = -cambioY;
 }

 private void dibujar() {
 Graphics papel = panel.getGraphics();
 papel.setColor(Color.red);
 papel.fillOval(x, y, diámetro, diámetro);
 }

 private void eliminar() {
 Graphics papel = panel.getGraphics();
 papel.setColor(Color.white);
 papel.fillOval (x, y, diámetro, diámetro);
 }
}
```

Para llamar a un método desde el interior del objeto, tenemos que hacer lo siguiente:

```
mover();
```

proporcionando el nombre del método y sus parámetros de la manera usual. Si en realidad queremos enfatizar cuál objeto se está usando, podríamos escribir el siguiente código equivalente:

```
this.mover();
```

al utilizar la palabra clave `this` estamos indicando que es el objeto actual.

Dependiendo de su tamaño y complejidad, una clase podría tener varios métodos `private`. Su propósito es clarificar y simplificar el funcionamiento de la clase.

## Reglas de alcance

En la programación, el término *accesibilidad* (algunas veces conocido como *reglas de alcance* o *visibilidad*) se refiere a las reglas para acceder a las variables y los métodos. Para los humanos, las reglas de accesibilidad son como la regla que establece que en Australia debemos conducir por la izquierda, o la regla que establece que sólo podemos entrar a la casa de alguien por la puerta delantera. En un programa, el compilador se encarga de hacer cumplir al pie de la letra las reglas de este tipo, para evitar un acceso deliberado o erróneo a la información protegida. Las reglas de accesibilidad restringen al programador, pero le ayudan a organizar un programa en forma clara y lógica. Las reglas de accesibilidad asociadas con las clases y los métodos permiten al programador encapsular las variables y los métodos en forma conveniente.

El programador puede describir cada variable y método como `public` o `private`. Dentro de una clase, cualquier instrucción en cualquier parte de la clase puede invocar a cualquier método, ya sea `public` o `private`. Además, cualquier instrucción puede hacer referencia a cualquier variable de instancia. La excepción es que las variables locales (las que se declaran dentro de un método) sólo pueden ser utilizadas por las instrucciones que están dentro del método.

Cuando una clase hace referencia a otra, sólo aquellos métodos y variables etiquetados como `public` se pueden utilizar fuera de la clase. Todos los demás elementos son inaccesibles. Es una buena práctica de diseño minimizar el número de métodos que sean `public`, restringiéndolos de manera que sólo se ofrezcan los servicios de la clase. También es buena práctica nunca (o muy pocas veces) hacer las variables `public`. Si debemos inspeccionar o modificar una variable, debe haber métodos para realizar este trabajo.

En resumen, una variable o un método dentro de una clase se pueden describir como:

1. `public` – se puede utilizar en cualquier parte (desde el interior de la clase o desde cualquier otra clase).
2. `private` – se puede utilizar sólo dentro de la clase.

Además, sólo se puede acceder a las variables locales (las variables que se declaran dentro de un método) desde el interior del método.

Las clases se etiquetan como `public` para poder usarlas en donde sea posible. Los constructores se etiquetan como `public` debido a que necesitamos llamarlos desde el exterior de la clase.

En el capítulo 10 volveremos a ver las reglas de alcance, cuando estudiemos el tema de la herencia.

## Operaciones sobre los objetos

Muchos de los objetos que se utilizan en los programas de Java se deben declarar como instancias de clases, pero algunos no. Las variables que se declaran como `int`, `boolean` y `double` se llaman tipos *primitivos*. Vienen incluidos como parte del lenguaje Java. Mientras que los nombres de las clases por lo general empiezan con letra mayúscula, los nombres de los tipos primitivos empiezan con minúscula. Al declarar una de estas variables la podemos usar de inmediato. Por ejemplo:

```
int número;
```

declara la variable `número` y la crea a la vez. Por el contrario, al crear cualquier otro objeto tenemos que utilizar de manera explícita la palabra clave `new`. Por ejemplo:

```
Globo globo = new Globo(10, 20, 50);
```

Entonces, las variables en Java pueden ser:

1. Tipos primitivos tales como `int`, `boolean` y `double`.
2. Objetos creados de manera explícita a partir de clases, mediante `new`.

Las variables que se declaran de un tipo primitivo incluyen toda una diversidad de cosas que podemos hacer con ellas. Por ejemplo, con las variables de tipo `int` podemos:

- Declarar variables.
- Asignar valores mediante `=`.
- Realizar operaciones aritméticas.
- Comparar mediante `==`, `<`, etc.
- Usarlas como parámetro o como valor de retorno.

No necesariamente podemos hacer todas estas cosas con los objetos. Muchas de las cosas que un programa de Java utiliza son objetos pero, como hemos visto, no todo es un objeto. Y es tentador suponer que sea posible utilizar todas estas operaciones con cualquier objeto, pero no es el caso. ¿Qué podemos hacer con un objeto? La respuesta es que al escribir una clase, definimos el conjunto de operaciones que se pueden realizar sobre los objetos de ese tipo. Por ejemplo, con la clase `Globo` hemos definido las operaciones `cambiarTamaño`, `mover` y `mostrar`. No debe suponer que podemos hacer cualquier otra cosa con un globo. Sin embargo, puede suponer con confianza que puede hacer lo siguiente con cada objeto:

- Crearlo mediante `new`.
- Usarlo como parámetro y como valor de retorno.
- Asignarlo a una variable de la misma clase mediante `=`.
- Usar los métodos que se proporcionan como parte de su clase.

## PRÁCTICA DE AUTOEVALUACIÓN

- 9.7 Sugiera una lista de operaciones que se puedan realizar con un objeto de la clase `Globo` y mencione ejemplos de cómo usarlas.

### ● Destrucción de objetos

Ya vimos cómo crear objetos mediante la poderosa palabra `new`. Pero, ¿cómo mueren? Una respuesta obvia y acertada es que esto ocurre cuando el programa deja de ejecutarse. También pueden morir cuando el programa deja de utilizarlos. Por ejemplo, si hacemos lo siguiente para crear un nuevo objeto:

```
Globo globo;
globo = new Globo(20, 100, 100);
```

y después hacemos lo siguiente:

```
globo = new Globo(40, 200, 200);
```

lo que ocurre es que el primer objeto creado con `new` tuvo una vida muy corta, ya que murió cuando el programa dejó de tener conocimiento de él y su valor fue usurpado por el objeto más reciente.

Cuando se destruye un objeto, se reclama la memoria que se utilizaba para almacenar los valores de sus variables y cualquier otro recurso para que el sistema en tiempo de ejecución la utilice en otros procesos. A esto se le conoce como *recolección de basura*. En Java la recolección de basura es automática (en algunos otros lenguajes como C++ no es automática, por lo que el programador tiene que llevar cuenta de los objetos que ya no son necesarios).

Por último, podemos destruir un objeto al asignarle el valor `null`; por ejemplo:

```
globo = null;
```

La palabra clave `null` de Java describe a un objeto no existente (no instanciado).

### ● Métodos static

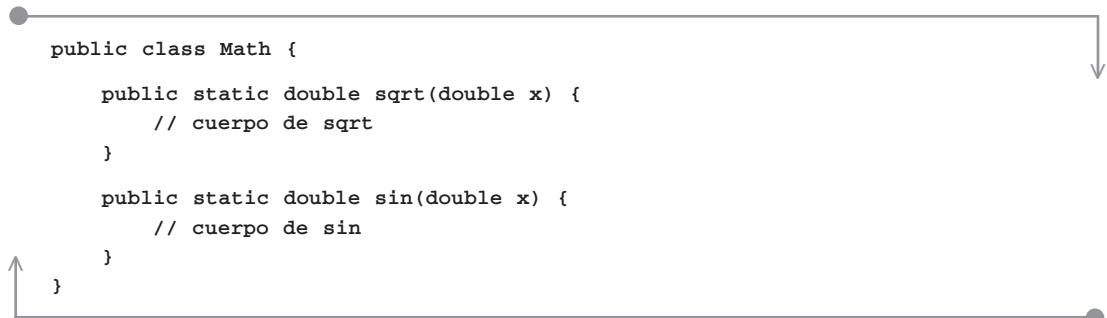
Algunos métodos no necesitan un objeto para trabajar. La función matemática de raíz cuadrada es un ejemplo. Los métodos matemáticos como la raíz cuadrada (`sqrt`) y el seno de un ángulo (`sin`) se proporcionan dentro de una clase de biblioteca llamada `Math`. Para usarlas en un programa escribimos instrucciones como:

```
double x, y;
x = Math.sqrt(y);
```

En esta instrucción hay dos variables `double` llamadas `x` y `y`, pero no hay objetos. Tenga en cuenta que `Math` es el nombre de una clase y no un objeto. El método de raíz cuadrada `sqrt` actúa sobre su parámetro `y`. La pregunta es: si `sqrt` no es método de un objeto, entonces ¿qué es? La respuesta es que los métodos como éste forman parte de una clase, pero se describen como `static`. Al uti-

lizar uno de estos métodos debemos anteponer a su nombre el nombre de la clase a la que pertenece (en vez del nombre de un objeto).

La clase **Math** tiene la siguiente estructura (el código que se muestra está incompleto). Los métodos se etiquetan como **static**:



El método **parseInt** es otro ejemplo de un método **static**, el cual se encuentra dentro de la clase **Integer**. El método **main** que aparece en el encabezado de todas las aplicaciones de Java también es un método **static**.

¿Cuál es la finalidad de los métodos **static**? En la POO todo se escribe como parte de una clase; nada existe que no esté dentro de las clases. Si pensamos en la clase **Globo**, ésta contiene variables **private** como **x** y **y** que registran el estado de un objeto. Pero algunos métodos, como **sqrt**, no involucran un estado. Sin embargo, los métodos independientes como **sqrt** que obviamente no forman parte de una clase deben obedecer la regla central de la POO: tienen que formar parte de una clase. He aquí la razón de los métodos **static**. Es común que los programadores utilicen los métodos **static** de biblioteca, pero es muy poco usual que los programadores novatos los escriban.

A los métodos **static** también se les denomina métodos de *clase*, ya que pertenecen a la clase y no a un objeto creado a partir de la clase.

## PRÁCTICA DE AUTOEVALUACIÓN

- 9.8** El método **static** llamado **max**, que está dentro de la clase **Math**, encuentra el máximo de sus dos parámetros **int**. Escriba un ejemplo de invocación a **max**.

### ● Variables **static**

Una variable que se declara en el encabezado de una clase se puede describir como **estática**, o **static**. Esto significa que pertenece a la clase y no a los objetos individuales que se crean como instancias de esa clase.

Como ejemplo, la clase **Math** contiene una variable **estática**, la constante matemática pi. Esta variable se referencia como **Math.PI**, en donde se antepone de nuevo el nombre de la clase al nombre de la variable. Es muy inusual en la POO que haya valores de datos públicos como éste, ya que por lo general las variables se etiquetan como **private** en virtud del ocultamiento de la informa-

ción. El acceso a las constantes matemáticas es una excepción a esta regla general. La variable `PI` se declara de la siguiente forma dentro de la clase `Math`:

```
public class Math {
 public final static double PI = 3.142;
 // resto de la clase Math
}
```

(con la excepción de que al valor de `pi` se le asigne una mayor precisión).

La clase `Color` contiene otros ejemplos de variables `static`. Esta clase provee variables que se referencian como `Color.black`, `Color.white`, etcétera.

La descripción `static` no significa que una variable `static` no se pueda modificar. Significa que, a diferencia de las variables no estáticas, no se crea una copia de la variable cuando se crea un objeto a partir de la clase. La descripción `static` implica unicidad; es decir, sólo hay una copia de esta variable para toda la clase, en vez de una copia para cada instancia de la clase.

A las variables `static` se les conoce algunas veces como variables de clase, pero no son lo mismo que las variables a nivel de clase.

## Fundamentos de programación

La POO trata acerca de cómo construir programas a partir de objetos. Un *objeto* es una combinación de ciertos datos (variables) y ciertas acciones (métodos) que desempeñan una función útil en un programa. El programador diseña un objeto de manera que los datos y las acciones estén muy relacionados entre sí, en vez de agruparlos al azar.

Al igual que en la mayoría de los lenguajes de POO, en Java no es posible escribir instrucciones que describan a un objeto en forma directa. En vez de ello, el lenguaje hace que el programador defina a todos los objetos de la misma clase. Por ejemplo, si necesitamos un objeto botón creamos una instancia de la clase `JButton`. Si necesitamos un segundo botón, creamos una segunda instancia de esta misma clase. La descripción de la estructura de todos los posibles botones se llama *clase*. Una clase es la plantilla o el plan maestro para fabricar cualquier número de objetos; una clase es la generalización de un objeto.

La idea de clases es común en la mayoría de las actividades de diseño. Por lo general, antes de construir algo real hay que crear un diseño para el objeto. Esto se aplica en el diseño automotriz, la arquitectura, la construcción —incluso en las bellas artes. Primero se bosqueja una especie de plano, a menudo en papel y algunas veces en la computadora. Comúnmente se le denomina plano de construcción. Dicho diseño especifica por completo el objeto deseado, de manera que si el diseñador es arrollado por un autobús alguien más pueda llevar a cabo la construcción del objeto. Una vez diseñado, se pueden construir varios objetos idénticos; piense en automóviles, libros o computadoras. De esta manera, el diseño especifica la composición de uno o cualquier cantidad de objetos. Lo mismo se aplica en la POO: una clase es el plano para cualquier número de objetos idénticos. Una vez que especificamos una clase, podemos construir cualquier número de objetos con el mismo comportamiento.

Si analizamos de nuevo la clase de biblioteca `JButton`, lo que tenemos es la descripción de la apariencia de cada objeto botón. Los botones sólo difieren en sus propiedades individuales, como sus posiciones en la pantalla. Por ello, en la POO una clase es la especificación para cualquier cantidad de objetos que son iguales. Una vez que se describe una clase, se puede construir un

objeto específico creando una *instancia* de esa clase. Es algo así como decir que hemos tenido una instancia de gripe en el hogar. O que este Ford Modelo T es una instancia del diseño del Ford Modelo T. Su propia cuenta bancaria es una instancia de la clase cuenta bancaria.

Un objeto constituye un agrupamiento lógico de variables y métodos. Forma un módulo independiente que se puede utilizar y entender con facilidad. El principio del ocultamiento de información o encapsulamiento implica que los usuarios de un objeto tienen una vista restringida del mismo. Un objeto proporciona un conjunto de servicios en forma de métodos **public** que otros pueden utilizar. El resto del objeto, sus variables y las instrucciones que implementan a los métodos están ocultos. Esto mejora la abstracción y la modularidad.

En computación a una clase se le denomina algunas veces *tipo de datos abstracto* (ADT). Un tipo de datos es algo así como una variable, por ejemplo **int**, **double** o **boolean**. Estos tipos primitivos son tipos integrados en el lenguaje Java y están disponibles para utilizarlos de inmediato. Hay un conjunto de operaciones asociado a cada uno de estos tipos. Por ejemplo, con un **int** podemos realizar operaciones de asignación, suma, resta, etcétera. La clase **Globo** que vimos antes es un ejemplo de ADT. Define ciertos datos (variables), junto con una colección de operaciones (métodos) que pueden realizar operaciones con los datos. La clase presenta una abstracción de un globo; los detalles concretos de la implementación están ocultos.

Ahora podemos entender por completo la estructura general de un programa. Todo programa de Java tiene un encabezado similar al siguiente (pero con el nombre de la clase apropiada):

```
public class UsarGlobo
```

Ésta es una descripción de una clase llamada **UsarGlobo** ya que, como todo lo demás en Java, un programa es una clase. Al iniciar un programa se hace una llamada al método **main** estático (**static**).

## Errores comunes de programación

Algunas veces los principiantes quieren codificar un objeto directamente. Esto no es posible; debemos declarar una clase y después crear una instancia de esa clase.

No olvide inicializar las variables de instancia de manera explícita, por medio de un método constructor o como parte de la misma declaración; además, no se confie de la inicialización predeterminada de Java.

Si usted declara:

```
Globo globo;
```

y después ejecuta la siguiente línea de código:

```
globo.mostrar(papel);
```

su programa terminará con un mensaje de error indicando que hay una excepción de apuntador a **null**. Esto se debe a que declaró un objeto pero no lo creó (con **new**). El objeto **globo** no existe. Dicho en forma más precisa, tiene el valor **null** —que significa lo mismo. En la programación elemental casi nunca es necesario utilizar el valor **null**, a menos que olvidemos utilizar **new**.

## Secretos de codificación

- Una clase tiene la siguiente estructura:

```
public class NombreClase {
 // declaraciones de las variables de instancia
 // declaraciones de los métodos
}
```

- Las variables y los métodos se pueden describir como **public** o **private**.
- Uno o más de los métodos en una clase pueden tener el mismo nombre que la clase. Se puede llamar a cualquiera de estos métodos constructores (con los parámetros apropiados) para inicializar el objeto al momento de crearlo.
- La declaración de un método **public** tiene la siguiente estructura:

```
public void nombreMétodo(parámetros) {
 // cuerpo
}
```

- Un método estático lleva el prefijo **static** en su encabezado.
- Para llamar a un método **static** de una clase:

```
NombreClase.nombreMétodo(parámetros);
```

## Nuevos elementos del lenguaje

- **class** – aparece en el encabezado de una clase.
- **public** – la descripción de una variable o un método que se pueden utilizar desde cualquier parte.
- **private** – la descripción de una variable o un método que sólo se pueden utilizar dentro de la clase.
- **new** – se utiliza para crear una nueva instancia de una clase (un nuevo objeto).
- **this** – el nombre del objeto actual.
- **null** – el nombre de un objeto que no existe.
- **static** – la descripción que se adjunta a una variable o un método que pertenece a una clase como un todo y no a una instancia creada como un objeto a partir de la clase.

## Resumen

- Un objeto es una colección de datos junto con las acciones (métodos) asociadas que pueden actuar sobre esos datos. Los programas de Java se construyen como una variedad de objetos.
- Una clase es la descripción de cualquier número de objetos.
- Por lo general, una clase consiste en declaraciones de variables privadas, seguidas de algunos métodos públicos.
- Los elementos de una clase se pueden declarar como `private` o `public`. Un elemento `private` sólo se puede utilizar dentro de la clase. Un elemento `public` se puede utilizar en cualquier parte (dentro o fuera de la clase). Al diseñar un programa de Java se evitan las variables `public` para mejorar el ocultamiento de información.
- El programador puede escribir métodos para inicializar un objeto al crearlo. A estos métodos se les denomina constructores y tienen el mismo nombre que la clase.
- La descripción `static` significa que la variable o el método pertenecen a la clase y no a objetos específicos.

## Ejercicios

- 9.1 Globos** Agregue varios datos más a la clase `Globo`: una variable `String` que guarde el nombre del globo y una variable `color` que describa su color. Agregue código para inicializar estos valores mediante un método constructor. Agregue el código para mostrar el balón de color y su nombre.  
Mejore el programa del globo con botones que muevan el globo a la izquierda, a la derecha, hacia arriba y hacia abajo.
- 9.2 Termómetro** Algunos termómetros registran las temperaturas máxima y mínima que se han alcanzado. Escriba un programa que simule un termómetro mediante un control deslizable y que muestre en campos de texto los valores máximo y mínimo a los que se ajustó el control deslizable. Escriba como una clase separada la pieza de programa que recuerde los valores mayor y menor, y que compare nuevos valores. Esta clase debe tener los métodos `setNuevoValor`, `getMenorValor` y `getMayorValor`.
- 9.3 Cuenta bancaria** Escriba un programa que simule una cuenta bancaria. Un botón permite hacer un depósito en la cuenta. El monto se introduce en un campo de texto. Un segundo botón permite hacer un retiro. El monto (el saldo) y el estado de la cuenta se deben mostrar en forma continua: el estado puede ser OK o sobregiro. Cree una clase llamada `Cuenta` para representar cuentas bancarias. Debe tener los métodos `depositar` y `retirar`, `getSaldoActual` y `setSaldoActual`.
- 9.4 Tanteador** Diseñe y escriba una clase que actúe como tanteador para un juego de computadora. Debe mantener un solo entero: la puntuación. Además debe proporcionar un método para inicializar la puntuación en cero, un método para incrementar la puntuación, un método para reducir la puntuación

y otro método para devolver la puntuación. Escriba un programa para crear un solo objeto y utilizarlo. Siempre debe aparecer la puntuación actual en pantalla, dentro de un campo de texto. Debe haber botones para incrementar, reducir e inicializar la puntuación con base en una cantidad introducida en un campo de texto.

- 9.5 Dados** Diseñe y escriba una clase que actúe como un dado, el cual se puede lanzar para obtener un valor del 1 al 6. Escriba la clase de manera que en un principio se obtenga el valor 6. Escriba un programa para crear un objeto dado y utilizarlo. La pantalla debe mostrar un botón que al oprimirlo haga que se lance el dado y se muestre su valor.

Después modifique la clase dado de manera que proporcione un valor que sea un punto mayor al que tenía la última vez que se lanzó; por ejemplo, 4 cuando era un 3. Cuando el último valor sea 6, el nuevo valor será 1.

Luego modifique la clase para que utilice el generador de números aleatorios de la biblioteca.

Algunos juegos como el backgammon necesitan dos dados. Escriba instrucciones de Java para crear dos instancias del objeto dado, lanzar los dados y mostrar los resultados.

- 9.6 Generador de números aleatorios** Escriba su propio generador de números aleatorios como una clase que utilice una fórmula para obtener el siguiente número pseudoaleatorio a partir del anterior. Un programa de números aleatorios funciona empezando con cierto valor de “semilla”. A partir de ese momento, el número aleatorio actual se utiliza como base para obtener el siguiente número, para lo cual realizamos ciertos cálculos con el número actual para convertirlo en algún otro número (aparentemente aleatorio). Una buena fórmula que podemos usar para los enteros es:

```
siguienteA = ((antiguoA * 25173) + 13849) % 65536;
```

esta fórmula produce números en el rango de 0 a 65,535. Los números específicos en esta fórmula han demostrado producir buenos resultados tipo aleatorios.

Al principio haga que el valor de semilla sea 1. Después, en un programa más sofisticado, obtenga la parte de los milisegundos del tiempo mediante el uso de la clase de biblioteca **Calendar** (vea el apéndice A) para que actúe como semilla.

- 9.7 Estacionamiento** Un programa provee dos botones. El asistente del estacionamiento hace clic en un botón cuando un automóvil entra al lote y presiona el otro botón cuando sale un automóvil. Si un automóvil intenta entrar cuando el lote ya está lleno, se muestra una advertencia en un panel de opción.

Implemente el conteo de automóviles y sus operaciones como una clase. Debe proveer un método llamado **entrar**, el cual debe incrementar el conteo; además, debe incluir un método llamado **salir**, que disminuya el conteo. Un tercer método (llamado **lleno**) debe devolver un valor **boolean** que especifique si el lote está lleno o no.

- 9.8 Números complejos** Escriba una clase llamada **Complejo** para representar números complejos (junto con sus operaciones). Un número complejo consta de dos partes: una parte real (un **double**) y una parte imaginaria (un **double**). El método constructor debe crear un nuevo número complejo mediante los valores **double** que se proporcionen como parámetros, de la siguiente forma:

```
Complejo c = new Complejo(1.0, 2.0);
```

Escriba los métodos **getReal** y **getImaginaria** para obtener la parte real y la parte imaginaria de un número complejo, las cuales se deben utilizar de la siguiente manera:

```
double x = c.getReal();
```

Escriba un método llamado **suma** para sumar dos números complejos y devolver su suma. La parte real es la suma de las dos partes reales. La parte imaginaria es la suma de las dos partes imaginarias. Una llamada al método sería así:

```
Complejo c = c1.suma(c2);
```

Utilice dos campos de texto para introducir los valores de **c1** y haga lo mismo para **c2**. También muestre los valores de **c** en dos campos de texto.

Escriba un método llamado **prod** para calcular el producto de dos números complejos. Si un número tiene los componentes  $x_1$  y  $y_1$ , y el segundo número tiene los componentes  $x_2$  y  $y_2$ :

La parte real del producto =  $x_1 \times x_2 - y_1 \times y_2$

La parte imaginaria del producto =  $x_1 \times y_2 + x_2 \times y_1$

## Respuestas a las prácticas de autoevaluación

- 9.1    `private Color color;`
- 9.2    `public void moverArriba(int cantidad) {`  
          `coordY = coordY - cantidad;`  
          `}`
- 9.3    `public void cambiarColor(Color nuevoColor) {`  
          `color = nuevoColor;`  
          `}`
- 9.4    `public void mostrar(Graphics papel) {`  
          `papel.setColor(color);`  
          `papel.drawOval(x, y, diámetro, diámetro);`  
          `}`
- 9.5    `public int getY() {`  
          `return y;`  
          `}`
- 9.6    `public Globo(int diámetroInicial) {`  
          `diámetro = diámetroInicial;`  
          `}`
- 9.7    Los métodos son: `cambiarColor`, `moverIzquierda`, `moverDerecha`, `cambiarTamaño`, `mostrar`, `getX`, `getY`.  
Algunos ejemplos son:  
  

```
globo.cambiarColor(Color.red);
globo.moverIzquierda(20);
globo.moverDerecha(50);
globo.cambiarTamaño(10);
globo.mostrar(papel);
int x = globo.getX();
int y = globo.getY();
```
- 9.8    `int x;`  
          `x = Math.max(7, 8);`

# CAPÍTULO 10



## Herencia

En este capítulo conoceremos cómo:

- Crear una nueva clase a partir de una clase existente mediante la herencia.
- Aclarar variables y métodos como **protected** y en qué situaciones utilizarlos.
- Utilizar la redefinición y en qué situaciones aplicarla.
- Dibujar un diagrama de clases que describa la herencia.
- Utilizar la palabra clave **super**.
- Escribir constructores para las subclases.
- Emplear la palabra clave **final**.
- Utilizar clases abstractas y la palabra clave **abstract**.

### ● Introducción

Los programas se crean a partir de objetos, los cuales son instancias de clases. Algunas clases están en la biblioteca de Java y otras las escribe el programador. Cuando empezamos a escribir un nuevo programa, buscamos clases útiles en la biblioteca y vemos las clases que hemos escrito en el pasado. Esta metodología OO para la programación significa que en vez de empezar los programas desde cero, nos basamos en el trabajo anterior. Es común encontrar una clase que parezca útil y que haga casi todo lo que necesitamos, pero que no sea exactamente lo que queremos. La herencia constituye una manera de resolver este problema. Con la herencia podemos usar una clase existente como la base para crear una clase modificada.

Considere la siguiente analogía. Suponga que desea comprar un automóvil nuevo y va a una agencia en donde hay una variedad de automóviles producidos en masa. A usted le gusta uno en especial, pero no tiene esa característica especial que está buscando. Al igual que la descripción de

una clase, el automóvil se fabricó a partir de planos que describen a muchos automóviles idénticos. Si estuviera disponible la herencia, usted podría especificar un automóvil que tuviera todas las características del automóvil producido en masa, pero con las características o cambios adicionales que usted requiere.

## Cómo usar la herencia

Vamos a empezar con una clase similar a la que ya hemos usado varias veces en el libro. Es una clase para representar a una esfera. Una esfera tiene un radio y una posición en el espacio. Al mostrar una esfera en la pantalla, debe aparecer como un círculo (el método para mostrar una esfera simplemente invoca al método de biblioteca `drawOval`). El diámetro de la esfera está fijo en 20 píxeles. Sólo hemos modelado las coordenadas *x* y *y* de una esfera (*y* no la coordenada *z*) debido a que vamos a mostrar una representación bidimensional en la pantalla.

He aquí la descripción para la clase `Esfera`:

```
import java.awt.*;
public class Esfera {
 protected int x = 100, y = 100;
 public void setX(int nuevaX) {
 x = nuevaX;
 }
 public void setY(int nuevaY) {
 y = nuevaY;
 }
 public void mostrar(Graphics papel) {
 papel.drawOval(x, y, 20, 20);
 }
}
```

Cabe mencionar que hay varios elementos nuevos en este programa, incluyendo la palabra clave `protected`. Esto se debe a que escribimos la clase de tal forma que se pueda utilizar para la herencia. En el transcurso de este capítulo veremos lo que significan estos nuevos elementos.

Vamos a suponer que alguien escribió y probó esta clase, y la puso a disposición de otros para que la utilicen. Ahora vamos a escribir un nuevo programa y necesitamos una clase muy parecida a ésta, sólo que para describir burbujas. Esta nueva clase llamada `Burbuja` nos permitirá hacer cosas adicionales: modificar el tamaño de una burbuja y moverla en sentido vertical. La limitación de la clase `Esfera` es que describe objetos que no se mueven y cuyo tamaño no se puede cambiar. Primero necesitamos un método adicional que nos permita establecer un nuevo valor para el radio de la burbuja. Podemos hacer esto sin alterar la clase existente; para ello debemos escribir una clase diferente que utilice el código que ya se encuentra en la clase `Esfera`. Decimos que la nueva clase hereda las variables y los métodos de la clase anterior. La nueva clase se convierte en una subclase de la anterior. A la clase anterior se le llama la superclase de la nueva clase. He aquí cómo podemos escribir la nueva clase:

```

import java.awt.*;
public class Burbuja extends Esfera {
 protected int radio = 10;
 public void setTamaño(int tamaño) {
 radio = tamaño;
 }
 public void mostrar(Graphics papel) {
 papel.drawOval(x, y, 2 * radio, 2 * radio);
 }
}

```

Esta nueva clase tiene el nombre **Burbuja**. La palabra clave **extends** y la mención de la clase **Esfera** indican que **Burbuja** hereda de la clase **Esfera**, o decimos que **Burbuja** es una subclase de **Esfera**. Esto significa que **Burbuja** hereda todos los elementos que no estén descritos como **private** dentro de la clase **Esfera**. En las siguientes secciones exploraremos las otras características de esta clase.

### **protected**

Cuando usamos la herencia, **private** es un término demasiado privado y **public** es demasiado público. Si una clase necesita dar a sus subclases acceso a ciertas variables o métodos específicos, pero debe evitar que otras clases accedan a éstos, puede etiquetarlos como **protected**. En la analogía de una familia, una madre permite a sus descendientes utilizar las llaves de su automóvil, pero a nadie más.

Volviendo a la clase **Esfera**, necesitamos variables para describir las coordenadas. Podríamos escribir lo siguiente:

```
private int x, y;
```

Ésta es una decisión acertada, pero debe haber una mejor idea. Podría darse el caso de que en el futuro alguien escribiera una clase que heredara de esta clase y proporcionara un método adicional para mover una esfera. Este método necesitaría acceso a las variables **x** y **y**, que por desgracia son inaccesibles ya que se etiquetaron como **private**. Por lo tanto, para anticiparnos a este posible uso en el futuro, podríamos decidir etiquetarlas mejor como **protected**:

```
protected int x, y;
```

Ahora esta declaración protege a las variables de manera que otras clases no puedan hacer mal uso de ellas, pero permite el acceso a ciertas clases privilegiadas: las subclases. El mismo principio se aplica a los métodos.

Suponga que declaramos a **x** y **y private**, como lo teníamos planeado en un principio. La consecuencia es que hubiera sido imposible reutilizar la clase mediante la herencia. La única opción sería editar la clase y reemplazar la descripción **private** por **protected** para esos elementos específicos. Pero esto viola uno de los principios de la POO: nunca modificar una clase existente que haya sido probada y utilizada. Por ende, al escribir una clase debemos esforzarnos para tener en cuenta los posibles usos de la clase en un futuro. El programador que escribe una clase siempre lo hace con la

esperanza de que alguien la reutilice y la extienda. El uso cuidadoso de `protected` en vez de `public` o `private` nos puede ayudar a hacer que una clase sea más atractiva para la herencia. Éste es otro de los principios de la POO.

## Reglas de alcance

En resumen, los cuatro niveles de accesibilidad (reglas de alcance) de una variable o un método en una clase son:

1. `public` – se puede utilizar en cualquier parte. Como regla, cualquier método que ofrezca un servicio a los usuarios de una clase se debe etiquetar como `public`.
2. `protected` – se puede utilizar dentro de esta clase y desde cualquier subclase.
3. `private` – sólo se puede utilizar dentro de esta clase. Por lo general, las variables de instancia se deben declarar como `private` y algunas veces como `protected`.
4. Las variables locales (que se declaran dentro de un método) nunca podrán utilizarse fuera del método específico.

Entonces una clase puede tener un acceso adecuado, pero controlado, a su superclase inmediata y a las superclases que estén arriba de ésta en la jerarquía de clases, como si las clases formaran parte de la clase en sí. Si recurrimos a la analogía de la familia, es como poder gastar libremente el dinero de nuestra madre o de cualquiera de sus ancestros, siempre y cuando hayan puesto su dinero en una cuenta etiquetada como `public` o `protected`. Las personas fuera de la familia sólo pueden acceder al dinero `public`.

## Elementos adicionales

Una forma importante de construir una nueva clase a partir de otra es mediante la inclusión de variables y métodos adicionales.

Podemos ver que la clase `Burbuja` anterior declara una variable y un método adicionales:

```
protected int radio = 10;
public void setTamaño(int tamaño) {
 radio = tamaño;
}
```

La nueva variable es `radio`, adicional a las variables existentes (`x` y `y`) en `Esfera`. Por lo tanto, se extiende el número de variables.

La nueva clase también tiene el método `setTamaño` además de los que contiene `Esfera`.

## PRÁCTICA DE AUTOEVALUACIÓN

- 10.1** Un objeto pelota es como un objeto `Esfera`, sólo que con las características adicionales de poder moverse a la izquierda y a la derecha. Escriba una clase llamada `Pelota` que herede la clase `Esfera` pero proporcione los métodos adicionales `moverIzquierda` y `moverDerecha`.

## ● Redefinición (o sobreescritura)

Otra característica de la nueva clase **Burbuja** es una nueva versión del método **mostrar**:

```
public void mostrar(Graphics papel) {
 papel.drawOval(x, y, 2 * radio, 2 * radio);
}
```

Esto es necesario debido a que la nueva clase tiene un radio que se puede modificar, mientras que en la clase **Esfera** el radio estaba fijo. Esta nueva versión de **mostrar** en **Burbuja** sustituye a la versión de la clase **Esfera**. Decimos que la nueva versión *redefine* a la versión anterior.

No confunda la redefinición con la sobrecarga, que vimos en el capítulo 5 sobre los métodos:

- Sobrecargar significa escribir un método (en la misma clase) que tenga el mismo nombre, pero una lista de parámetros distinta.
- Redefinir significa escribir un método en una subclase que tenga el mismo nombre y los mismos parámetros que en la superclase.

En resumen, en la clase que hereda hemos:

- Creado una variable adicional.
- Creado un método adicional.
- Redefinido un método (proporcionamos un método que se va a utilizar en vez del método que ya existía).

Ahora veamos un resumen de lo que hemos logrado. Teníamos una clase llamada **Esfera**. Requeríamos una nueva clase llamada **Burbuja** que fuera similar a **Esfera**, pero necesitábamos herramientas adicionales. Por ende, para crear la nueva clase extendimos las herramientas de la clase que ya teníamos. Hemos explotado al máximo las características comunes entre las dos clases, y hemos evitado volver a escribir piezas del programa que ya existían. Ambas clases que escribimos (**Esfera** y **Burbuja**) están disponibles para que las utilicemos.

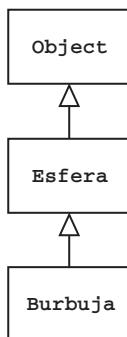
Como analogía con las familias humanas, la herencia significa que podemos gastar nuestro propio dinero y también el de nuestra madre.

Cabe mencionar que técnicamente es posible redefinir variables: declarar variables en una subclase que redefinan a las variables de la superclase. No hablaremos más sobre esto por dos buenas razones: la primera, nunca tendremos la necesidad de hacer esto; y la segunda, es muy mala práctica. Al crear una subclase a partir de una clase (heredar de ella), sólo tendremos que:

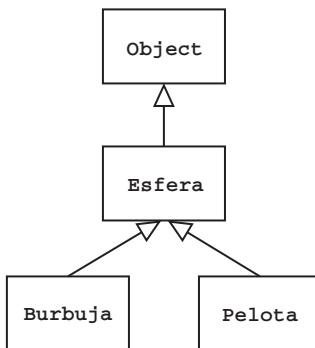
- Agregar métodos adicionales.
- Agregar variables adicionales.
- Redefinir métodos.

## ● Diagramas de clases

Una buena forma de visualizar la herencia es mediante un diagrama de clases, como se muestra en la figura 10.1. Esta figura nos muestra que **Burbuja** es una subclase de **Esfera**, la cual a su vez es una subclase de **Object**. Cada clase se muestra como un rectángulo. Una línea entre las clases muestra una relación de herencia. La flecha apunta de la subclase a la superclase.



**Figura 10.1** Diagrama de clases para las clases **Esfera** y **Burbuja**.



**Figura 10.2** Diagrama de clases que muestra una estructura tipo árbol.

Las clases de la biblioteca o las que escribe el programador encajan dentro de una jerarquía de clases. Si usted escribe una clase que empiece con el siguiente encabezado:

```
public class Esfera
```

que no tiene una superclase explícita, entonces es implícitamente una subclase de la clase **Object**. Por lo tanto, toda clase es implícita o explícitamente una subclase de **Object**.

La figura 10.2 muestra otro diagrama de clases en el que otra clase llamada **Pelota** también es una subclase de **Esfera**. Ahora el diagrama es una estructura tipo árbol, en la cual la raíz del árbol, **Object**, está en la parte superior. En general, un diagrama de clases es un árbol, parecido a un árbol genealógico, con la diferencia de que sólo aparece un parente.

## ● La herencia en acción

Al igual que la clase **Burbuja**, es común que una clase tenga una superclase, la que a su vez tiene una superclase y así sucesivamente, hasta llegar a la parte superior del árbol de herencia. No sólo se heredan los elementos **public** y **protected** de la superclase inmediata, sino también todas las variables y los métodos **public** y **protected** de todas las superclases en el árbol de herencia. Esto es similar al árbol genealógico: usted hereda de su madre, su abuela y así de manera sucesiva.

Suponga que creamos un objeto `burbuja` a partir de la clase `Burbuja`:

```
Burbuja burbuja = new Burbuja();
```

¿Qué ocurre si utilizamos el método `setX` de la siguiente manera?

```
burbuja.setX(200);
```

Aquí `burbuja` es un objeto de la clase `Burbuja`, pero `setX` no es un método de `Burbuja`, sino un método de una clase distinta llamada `Esfera`. Esto está bien, ya que todos los métodos etiquetados como `public` (y `protected`) dentro de la superclase inmediata (y de todas las superclases en la jerarquía de clases) están disponibles para una subclase. Y como `Burbuja` es una subclase de `Esfera`, `setX` está disponible para los objetos de la clase `Burbuja`.

La regla es que al utilizar un método, el sistema de Java primero busca en la clase del objeto para tratar de encontrar el método. Si no lo encuentra ahí, busca en la clase de la superclase inmediata. Si no lo puede encontrar ahí, busca en la clase de la superclase de la superclase, y así recorre toda la jerarquía de clases hasta encontrar un método con el nombre requerido. En la analogía familiar, usted hereda implícitamente de su abuela, de su bisabuela, y así sucesivamente.

El lenguaje Java permite que una clase herede sólo de una superclase inmediata. A esto se le conoce como herencia simple. En la analogía de la familia significa que usted puede heredar de su madre, pero no de su padre.

### ● `super`

Algunas veces una clase necesita llamar a un método de su superclase inmediata, o de alguna de las clases del nivel superior en el árbol. No hay problema con esto, ya que los métodos de todas las clases en los niveles superiores del árbol de herencia están disponibles, siempre y cuando estén etiquetados como `public` o `protected`. El único problema que puede surgir es cuando el método deseado de la superclase tiene el mismo nombre que un método de la clase actual (cuando se utiliza la redefinición o sobreescritura). Para corregir este problema hay que anteponer al nombre del método la palabra clave `super`. Por ejemplo, para llamar al método `mostrar` de una superclase use lo siguiente:

```
super.mostrar(papel);
```

En general esto es más ordenado y corto que duplicar instrucciones, y nos puede ayudar a que un programa sea más conciso al utilizar al máximo los métodos existentes.

### ● Constructores

En el capítulo 9 vimos sobre los constructores, a la hora de escribir clases. Los constructores nos permiten pasar parámetros a un objeto cuando lo creamos mediante la palabra reservada `new`. Un constructor es un método con el mismo nombre que la clase. Recuerde que:

- Si escribe una clase sin constructores, Java supone que sólo hay un constructor (con cero parámetros).

- Si escribe una clase con uno o más constructores con parámetros y también necesita un constructor sin parámetros, debe escribirlo en forma explícita.

Hay dos reglas relacionadas con la herencia y los constructores:

- Los constructores no se heredan.
- Una subclase debe llamar a uno de los constructores de la superclase.

Veremos cada una de estas reglas en su momento.

Los métodos constructores no se heredan. Ésta es una regla razonable; dice que un constructor está asociado con la inicialización de una clase específica y no de las subclases. Pero significa que si usted necesita uno o más constructores (como se da comúnmente el caso) en una subclase, tiene que escribirlos en forma explícita.

El constructor de una subclase debe llamar a uno de los constructores de la superclase. Además, ésta debe ser la primera acción del constructor. De nuevo, esto es razonable: la superclase se debe inicializar en forma apropiada y antes de que se creen e inicialicen nuevos elementos en la subclase. Si la primera instrucción en un constructor no es una llamada a un constructor de la superclase, entonces Java llama de manera automática al constructor sin parámetros de la superclase. Por ende, Java obliga al programador a llamar a uno de los constructores de la superclase.

Ahora veamos cómo funciona esto en la práctica. Por ejemplo, suponga que tenemos una clase con dos constructores:

```
import java.awt.*;
public class Globo {
 protected int x, y, radio;
 public Globo() {
 x = 10;
 y = 10;
 radio = 20;
 }
 public Globo(int xInicial, int yInicial,
 int radioInicial) {
 x = xInicial;
 y = yInicial;
 radio = radioInicial;
 }
 // resto de la clase
}
```

Si ahora escribimos una nueva clase llamada **GloboDiferente** que herede de la clase **Globo**, las opciones son:

1. No escribir un constructor en la subclase. Java supondrá que hay un constructor sin parámetros.

2. Escribir uno o más constructores sin llamadas a los constructores de la superclase.
3. Escribir uno o más constructores que llamen a un constructor apropiado de la superclase, utilizando `super`.

Para los programadores principiantes (y tal vez para los expertos también) probablemente sea mejor dejar las cosas en claro y escribir de manera explícita una llamada a un constructor de la superclase. Haremos esto en los siguientes ejemplos.

Veamos a continuación una subclase de `Globo` con un constructor que invoca al constructor sin parámetros de la superclase, mediante `super`:

```
public class GloboDiferente extends Globo {
 public GloboDiferente(int xInicial, int yInicial) {
 super();
 x = xInicial;
 y = yInicial;
 radio = 20;
 }
 // resto de la clase
}
```

y he aquí una subclase con un constructor que llama de manera explícita al segundo constructor de la superclase, de nuevo mediante `super`:

```
public class GloboModificado extends Globo {
 public GloboModificado(int xInicial, int yInicial,
 int radioInicial) {
 super(xInicial, yInicial, radioInicial);
 }
 // resto de la clase
}
```

## PRÁCTICA DE AUTOEVALUACIÓN

- 10.2** Una esfera coloreada es como una esfera, sólo que de cierto color. Escriba una nueva clase llamada `EsferaColoreada` que extienda la clase `Esfera` para proporcionar un color que se pueda establecer al momento de crear el globo. Para ello utilice un método constructor, de tal forma que su clase permita escribir lo siguiente:

```
EsferaColoreada esferaColoreada =
 new EsferaColoreada(Color.red);
```

## PRÁCTICA DE AUTOEVALUACIÓN

**10.3** ¿Cuál es el problema con la subclase en el siguiente código?

```
public class CuentaBancaria {

 protected int depósito;

 CuentaBancaria(int depósitoInicial) {
 // resto del constructor
 }

 // resto de la clase
}

public class CuentaMejorada extends CuentaBancaria {

 public CuentaMejorada() {
 depósito = 1000;
 }

 // resto de la clase
}
```

### final

Los procesos de heredar y redefinir, se enfocan en cambiar el comportamiento de las clases y los objetos. La herencia es muy poderosa, pero algunas veces es reconfortante que algunas cosas estén fijas y no se puedan modificar. Por ejemplo, es bueno saber exactamente qué es lo que hace `sqrt`, qué es lo que hace `drawLine`, etcétera. En la POO siempre existe el peligro de que alguien extienda las clases a las que éstas pertenecen y en consecuencia cambie lo que hacen. Esto podría ser por error o en un intento inadvertido de ser útil. Para evitar esto el programador puede describir un método como `final`. Esto significa que no se puede redefinir. La mayoría de los métodos de la biblioteca se describen como `final`. Esto significa que cada vez que los use, puede estar completamente seguro de lo que hacen.

Como hemos visto desde los primeros capítulos del libro, las variables también se pueden declarar como `final`. Esto significa también que no podemos cambiar sus valores. Son constantes. Por ejemplo:

```
final double cmPerInch = 2.54;
```

declara una variable cuyo valor no se puede alterar. Así, el prefijo `final` tiene el mismo significado sin importar que se utilice en una variable o en un método.

Podemos describir toda una clase como `final`, lo cual significa que no se pueden crear subclases a partir de ella. Además, todos sus métodos serán implícitamente `final`.

Hacer una clase o un método `final` es una decisión seria, ya que impide la herencia: una de las herramientas avanzadas de la POO.

## Clases abstractas

Considere un programa que mantiene formas gráficas de todos tipos y tamaños: círculos, rectángulos, cuadrados, triángulos, etc. Estas distintas formas, similares a las clases que ya hemos visto en este capítulo, tienen información en común: su posición, color y tamaño. Vamos a declarar una superclase llamada **Forma** que describa los datos comunes. Cada clase individual hereda esta información común. He aquí la forma de describir esta superclase común:

```
import java.awt.*;
public abstract class Forma {
 protected int x, y ;
 protected int tamaño;
 public void moverDerecha() {
 x = x + 10;
 }
 public abstract void mostrar(Graphics papel);
}
```

La clase para describir círculos hereda de la clase **Forma**, como se muestra a continuación:

```
import java.awt.*;
public class Círculo extends Forma {
 public void mostrar(Graphics papel) {
 papel.drawOval(x, y, tamaño, tamaño);
 }
}
```

Por lo tanto, al escribir la clase **Círculo** hemos aprovechado las herramientas que proporciona la clase **Forma**.

Es inútil tratar de crear un objeto a partir de la clase **Forma**, ya que está incompleta. Ésta es la razón por la cual se incluye la palabra clave **abstract** en el encabezado de la clase **Forma**; el compilador impedirá cualquier intento por crear una instancia de esta clase. Se proporciona el método **moverDerecha** y está completo, y es heredado por cualquier subclase. Pero el método **mostrar** es tan sólo un encabezado (sin cuerpo). Se describe con la palabra clave **abstract** para indicar que cualquier subclase debe proveer una implementación de este método. La clase **Forma** se denomina clase abstracta debido a que no existe como una clase completa, sino que se proporciona simplemente para usarse en la herencia.

Hay una regla razonable que establece que si una clase contiene métodos que sean **abstract**, la misma clase se debe etiquetar también como **abstract**.

Las clases abstractas nos permiten aprovechar las características comunes de las clases. Al declarar una clase como **abstract** el programador que la utiliza (mediante la herencia) se ve obligado a proveer los métodos faltantes. Por lo tanto, ésta es una forma mediante la cual el diseñador de una clase puede fomentar un diseño en particular.

Se utiliza el término *abstracto* ya que, a medida que buscamos cada vez más alto en la jerarquía de clases, éstas se vuelven cada vez más generales o abstractas. En el ejemplo anterior, la clase **Forma** es más abstracta y menos concreta que la clase **Círculo**. La superclase abstrae las características (como la posición y el tamaño en este ejemplo) que son comunes entre sus subclases. Es común en

los programas OO extensos descubrir que los primeros niveles de las jerarquías de clases constan de métodos abstractos. De manera similar en la biología, la abstracción se utiliza en clases como la de los mamíferos, que no existen (por sí solas) pero sirven como superclases abstractas para un diverso conjunto de subclases. Por ejemplo, nunca hemos visto un objeto mamífero, pero hemos visto una vaca, que es una instancia de una subclase de mamífero.

## PRÁCTICA DE AUTOEVALUACIÓN

**10.4** Escriba una clase llamada `Cuadrado` que utilice la clase abstracta `Forma` antes descrita.

## Principios de programación

Escribir programas como una colección de clases implica que los programas son modulares. Otro de los beneficios es que las partes de un programa se pueden reutilizar en otros programas. La herencia es otra forma en la que la POO provee el potencial de la reutilización. Algunas veces los programadores se ven tentados a reinventar la rueda: desean escribir nuevo software cuando simplemente podrían utilizar el software existente. Una de las razones para escribir nuevo software es que es divertido. El problema es que cada vez se está volviendo más complejo, por lo que simplemente no hay suficiente tiempo para escribirlo desde cero. Imagine tener que escribir el software para crear los componentes de GUI que proporcionan las bibliotecas de Java. Imagine tener que escribir una función matemática como `sqrt` cada vez que la necesite. Simplemente se llevaría demasiado tiempo. Por ende, una buena razón para reutilizar software es ahorrar tiempo. No sólo nos ahorraremos el tiempo para escribir el software sino también el tiempo para probarlo exhaustivamente, lo cual puede requerir aún más tiempo que el mero proceso de escribir el software. De aquí que la reutilización de clases tenga sentido.

Una razón por la que algunas veces los programadores no reutilizan el software es debido a que el software existente no hace exactamente lo que necesitan. Tal vez haga el 90% de lo que quieren, pero faltan ciertas características imprescindibles o algunas otras hacen las cosas de manera distinta. Una metodología sería modificar el software existente para que coincida con las nuevas necesidades. Sin embargo, ésta es una peligrosa estrategia ya que el proceso de modificar software es como estar en un campo de minas. El software es más quebradizo que suave; al tratar de modificarlo, se rompe. Al modificar el software es muy fácil introducir errores nuevos y sutiles, los cuales requieren de un extenso proceso de depuración y corrección. Ésta es una experiencia común, tanto que los programadores se muestran muy reacios a modificar el software.

Aquí es donde la POO entra en acción. En un programa OO podemos heredar el comportamiento de aquellas partes de cierto software que necesitamos, redefinir los (pocos) métodos de los cuales requerimos un comportamiento distinto, y agregar nuevos métodos que realicen cosas adicionales. A menudo podemos heredar la mayor parte de una clase y realizar sólo unos cuantos cambios que sean necesarios mediante la herencia. Sólo tenemos que probar las piezas nuevas, con la seguridad de que el resto ya ha sido probado. Entonces el problema de la reutilización queda resuelto. Podemos utilizar el software existente en forma segura. Mientras tanto, la clase original permanece intacta, confiable y utilizable.

La herencia es como ir a comprar un automóvil. Usted puede ver el automóvil que quiere, y es casi perfecto pero le gustaría realizar unas pequeñas modificaciones, como un color dis-



*Principios de programación (continúa)*

tinto o incluir la navegación por satélite. Con la herencia puede heredar el automóvil estándar y modificar las partes según lo requiera.

La POO implica basarse en el trabajo de otros. El programador OO procede de esta forma:

1. Aclara los requerimientos del programa.
2. Explora la biblioteca en busca de clases que realicen las funciones requeridas y las utiliza para obtener los resultados deseados.
3. Revisa las clases de otros programas que ha escrito y las utiliza de la manera apropiada.
4. Extiende las clases de biblioteca o sus propias clases mediante la herencia cuando es útil.
5. Escribe sus propias clases nuevas.

Esto explica por qué los programas OO son con frecuencia muy cortos: simplemente utilizan las clases de biblioteca o crean nuevas clases que heredan de las clases de biblioteca. Esta metodología requiere una inversión en tiempo: el programador necesita un conocimiento profundo de las bibliotecas. Esta idea de reutilizar software OO es tan poderosa que algunas personas piensan en la POO sólo de esta forma. Desde este punto de vista, la POO es el proceso de extender las clases de biblioteca de manera que cumplan los requerimientos de una aplicación específica.

Casi todos los programas de este libro utilizan la herencia. Cada programa empieza con una línea muy similar a la siguiente:

```
public class Bóveda extends JFrame
```

Esta línea indica que la clase **Bóveda** hereda características de la clase de biblioteca **JFrame**. Las herramientas de **JFrame** incluyen métodos para crear una ventana de GUI con los iconos usuales para cambiar el tamaño de la ventana y cerrarla. Extender la clase **JFrame** es la principal forma en la que los programas de este libro extienden las clases de biblioteca.

**Tenga cuidado:** algunas veces la herencia no es la técnica apropiada. La composición (utilizar las clases existentes sin modificarlas) es una alternativa que puede ser más conveniente en ocasiones. En el capítulo 18, sobre diseño, veremos esta cuestión con más detalle.

## Errores comunes de programación

Los programadores principiantes utilizan la herencia de una clase de biblioteca, la clase **JFrame**, desde su primer programa. Pero aprender a utilizar la herencia dentro de nuestras propias clases es algo que lleva tiempo y experiencia. Por lo general, sólo vale la pena usarla en programas extensos. No se preocupe si no utiliza la herencia durante sus primeros programas.

Es común confundir la sobrecarga con la redefinición:

- *Sobrecargar* significa escribir dos o más métodos en la misma clase con el mismo nombre (pero una lista distinta de parámetros).
- *Redefinir (sobreescribir)* significa escribir un método en una subclase para utilizarlo en vez del método de la superclase (o de una de las superclases por encima de ella en el árbol de herencia).

## Nuevos elementos del lenguaje

- **extends**: indica que esta clase hereda de otra clase.
- **protected**: la descripción de una variable o un método que se pueden utilizar dentro de la clase o de cualquier subclase (pero no desde ningún otro lado).
- **abstract**: la descripción de una clase abstracta que no se puede crear, es decir, que no es instanciable, sino que se proporciona sólo para utilizarse en la herencia.
- **abstract**: la descripción de un método que se proporciona sólo como encabezado y debe ser proporcionado por una subclase.
- **super**: el nombre de la superclase de una clase, la clase de la que hereda.
- **final**: describe a un método o una variable que no se puede redefinir.

## Resumen

Extender (heredar) las herramientas de una clase es una buena forma de utilizar las partes existentes de los programas (clases).

Una subclase hereda las herramientas de su superclase inmediata y de todas las superclases por encima de ella en el árbol de herencia.

Una clase sólo tiene una superclase inmediata (sólo puede heredar de una clase). A esto se le conoce como *herencia simple* en la jerga de la POO.

Una clase puede extender las herramientas de una clase existente si proporciona uno o más de los siguientes elementos:

- Métodos adicionales.
- Variables adicionales.
- Métodos que redefinen a (actúan en vez de) los métodos de la superclase.

Una variable o un método se pueden describir con uno de los siguientes tres tipos de acceso:

- **public**: se puede utilizar desde cualquier clase.
- **private**: se puede utilizar sólo dentro de esta clase.
- **protected**: se puede utilizar sólo dentro de esta clase y de cualquier subclase.

Un diagrama de clases es un árbol que muestra las relaciones de herencia.

Para hacer referencia al nombre de la superclase de una clase se utiliza la palabra clave **super**.

Una clase abstracta se describe como **abstract**. No se puede instanciar para producir un objeto, ya que está incompleta. Dicha clase provee variables y métodos útiles que las subclases pueden heredar.

## Ejercicios

- 10.1 Nave espacial** Escriba una clase llamada `NaveEspacial` que describa a una nave espacial. Se debe comportar de la misma forma que un objeto `Esfera`, sólo que se puede mover hacia arriba y hacia abajo. Recurra a la herencia para heredar de la clase `Esfera` que se muestra en el texto.

Dibuje un diagrama de clases para mostrar cómo se relacionan las distintas clases.

- 10.2 Fútbol** Escriba una clase llamada `Fútbol` que restrinja los movimientos de una pelota, de manera que la coordenada `x` deba ser mayor o igual a 0 y menor o igual a 200, lo que corresponde a la longitud de una cancha de fútbol. Recurra a la herencia para heredar de la clase `Pelota`.

- 10.3 El banco** Una clase describe cuentas bancarias y provee los métodos `abonarCuenta`, `cargarCuenta`, `calcularInterés` y `getSaldoActual`. Una nueva cuenta se crea con un nombre y un saldo inicial.

Hay dos tipos de cuentas: una cuenta regular y una cuenta dorada. La cuenta dorada produce un interés del 5%, mientras que la cuenta regular produce un interés del 6% menos un cargo fijo de \$100. Cada vez que se hace un retiro, se revisa una cuenta regular para ver si está sobregirada. El cliente con una cuenta dorada puede sobregirarse en forma indefinida.

Escriba clases que describan los dos tipos de cuentas y utilice una clase abstracta para describir las características comunes (para simplificar, asuma que las cantidades de dinero se guardan como valores `int`).

- 10.4 Formas** Escriba una clase abstracta llamada `Forma` para describir objetos gráficos bidimensionales (cuadrado, círculo, rectángulo, triángulo, etc.) que tengan las siguientes características. Todos los objetos usan variables `int` que especifican las coordenadas `x` y `y` de la esquina superior izquierda de un rectángulo circundante, además de variables `int` que describen la altura y la anchura del rectángulo. Todos los objetos comparten los mismos métodos `setX` y `setY` para establecer los valores de estas coordenadas. Todos los objetos comparten los métodos `setAnchura` y `setAltura` para establecer los valores de la anchura y la altura del objeto. Todos los objetos tienen un método llamado `getÁrea` que devuelve el área del objeto, junto con un método `mostrar` para mostrarlo en pantalla, pero estos métodos son distintos dependiendo del objeto específico.

Escriba una clase llamada `Rectángulo` que herede de la clase `Forma`.

## Respuestas a las prácticas de autoevaluación

- 10.1    `public class Pelota extends Esfera {  
 public void moverIzquierda(int cantidad) {  
 x = x - cantidad;  
 }  
 public void moverDerecha(int cantidad) {  
 x = x + cantidad;  
 }  
}`
- 10.2    `public class EsferaColoreada extends Esfera {  
 private Color color;  
 public EsferaColoreada(Color colorInicial) {  
 color = colorInicial;  
 }  
}`
- 10.3    El compilador encontrará una falla en la subclase. No hay una llamada explícita a un constructor de la superclase, por lo que Java tratará de llamar a un constructor sin parámetros de la superclase y no se ha escrito dicho método.
- 10.4    `import java.awt.*;  
  
public class Cuadrado extends Forma {  
  
 public void mostrar(Graphics papel) {  
 papel.drawRect(x, y, tamaño, tamaño);  
 }  
}`

# CAPÍTULO **11**



## Cálculos

En este capítulo conoceremos cómo:

- Utilizar las funciones de la biblioteca matemática.
- Aplicar formato a los números para mostrarlos de una manera conveniente.
- Llevar a cabo cálculos comerciales y científicos.

### ● Introducción

Ya vimos en el capítulo 4 cómo llevar a cabo cálculos simples. Este capítulo trata sobre cálculos más serios. Mejora la explicación anterior y reúne toda la información necesaria para escribir programas que lleven a cabo cálculos. Si no le interesan los programas que realizan cálculos numéricos, puede pasar al siguiente capítulo.

Los cálculos surgen en muchos programas, no sólo en los que realizan cálculos matemáticos, científicos o de ingeniería, sino también en los sistemas de información, la contabilidad y los pronósticos. En los gráficos, los cálculos son necesarios para escalar y desplazar imágenes en la pantalla.

En el capítulo 4 explicamos varias ideas importantes sobre los números y los cálculos. Tal vez quiera repasar ese capítulo antes de continuar. Estas ideas son:

- Declaración de variables como `int` o `double`.
- Entrada y salida de datos mediante campos de texto.
- Conversión entre las representaciones de cadena de los números y sus representaciones internas.
- Reglas de precedencia en las expresiones.
- Conversiones en expresiones que mezclan datos `int` y `double`.

## ● Funciones y constantes de la biblioteca matemática

Es común en los programas matemáticos, científicos o de ingeniería utilizar funciones como seno, coseno y logaritmo. En Java estas funciones se incluyen en una de las bibliotecas: la biblioteca `Math`. Para usar una de las funciones podemos escribir algo como lo siguiente:

```
x = Math.sqrt(y);
```

A continuación le mostraremos algunas de las funciones más utilizadas en la biblioteca `Math`. Cuando el parámetro es un ángulo, se debe expresar en radianes.

|                        |                                                                                    |
|------------------------|------------------------------------------------------------------------------------|
| <code>cos(x)</code>    | coseno del ángulo $x$ , en donde $x$ se expresa en radianes                        |
| <code>sin(x)</code>    | seno del ángulo $x$ , expresado en radianes                                        |
| <code>tan(x)</code>    | la tangente del ángulo $x$ , expresada en radianes                                 |
| <code>abs(x)</code>    | el valor absoluto de $x$ , que algunas veces se escribe como $ x $ en matemáticas. |
| <code>min(x, y)</code> | el menor de $x$ y $y$                                                              |
| <code>max(x, y)</code> | el mayor de $x$ y $y$                                                              |
| <code>log(x)</code>    | logaritmo natural de $x$ (en base e)                                               |
| <code>random()</code>  | provee un número seudoaleatorio en el rango de 0.0 a 0.999...                      |
| <code>sqrt(x)</code>   | la raíz cuadrada positiva de $x$                                                   |
| <code>pow(x, y)</code> | $x$ elevado a la potencia de $y$ , o $x^y$                                         |
| <code>exp(x)</code>    | $e^x$                                                                              |
| <code>round(x)</code>  | el número entero más cercano a $x$                                                 |

Al utilizar estos métodos, algunas veces debemos tener cuidado en cuanto al tipo de variables o literales que utilizamos como parámetros. Por ejemplo, el método `abs` puede recibir cualquier valor numérico, pero el método `cos` sólo puede recibir un número `double`.

Las constantes matemáticas  $\pi$  y  $e$  también están disponibles como constantes dentro de la biblioteca `Math`, así que podemos escribir lo siguiente:

```
double x, y;
x = Math.PI;
y = Math.E;
```

## ● Cómo aplicar formato a los números

Aplicar formato significa mostrar los números de una forma conveniente. Por ejemplo, no siempre requerimos el detalle de las posiciones decimales innecesarias. Si el valor `33.124765` representa el área de una habitación en metros cuadrados, entonces tal vez no sean necesarias todas las posiciones decimales, ya que por lo general sólo habría que mostrar el número como `33.12`.

Java cuenta con una variedad de herramientas para aplicar formato a los valores, pero aquí nos limitaremos a los casos más comunes en los que se aplicará formato a valores `int` y `double`. Los pasos son:

- Crear una instancia de la clase de biblioteca `DecimalFormat` y suministrarle un patrón como parámetro.
- Llamar al método `format` y suministrarle como parámetro el número al que se va a aplicar formato, para que devuelva el valor con formato como una cadena de texto.

Empezaremos con valores `int`. Suponga que tenemos el siguiente valor entero:

```
int i = 123;
```

Ya hemos utilizado antes el método `toString` en varias ocasiones para convertir un número. Por ejemplo:

```
campoTexto.setText(Integer.toString(i));
```

con esto obtenemos la siguiente cadena de texto:

```
123
```

Pero ahora utilizaremos el método `format` para aplicar formato y obtener el mismo resultado:

```
int i = 123;
DecimalFormat formato = new DecimalFormat("###");
campoTexto.setText(formato.format(i));
```

Con esto el campo de texto recibe el mismo resultado:

```
123
```

El carácter # dentro de un patrón (tal como el anterior) significa insertar un dígito. En este caso, si hay menos de tres dígitos, no se crea ningún carácter.

Cuando sabemos que un entero puede ser hasta de cinco caracteres, por ejemplo, y queremos alinear los números, podemos utilizar lo siguiente:

```
int i = 123;
DecimalFormat formato = new DecimalFormat("00000");
campoTexto.setText(formato.format(i));
```

con lo cual obtenemos:

```
00123
```

Esto aplica formato al número utilizando exactamente cinco dígitos alineados a la derecha y se rellena con ceros a la izquierda según sea necesario.

Para números más grandes, las comas pueden mejorar la legibilidad del número. Esto se ilustra en el siguiente código:

```
int i = 123456;
DecimalFormat formato = new DecimalFormat("###,###");
campoTexto.setText(formato.format(i));
```

Aquí, el carácter , indica que el formato debe colocar una coma en esta posición. Con esto obtenemos la siguiente cadena:

```
123,456
```

El formato tiende a ser más útil cuando se van a mostrar valores `double`. En este ejemplo:

```
double d = 12.34;
DecimalFormat formato = new DecimalFormat("##.##");
campoTexto.setText(formato.format(d));
```

el campo de texto recibe el siguiente valor:

`12.34`

Los dos caracteres `#` especifican dos dígitos después del punto decimal. A la izquierda del punto decimal se muestran los dígitos que sean necesarios para presentar todo el número. Pero por lo menos se muestra un dígito. A la derecha del punto decimal, el número se redondea si es necesario para ajustarlo a los dos dígitos especificados.

También se provee una herramienta para mostrar números `double` en notación científica, para lo cual utilizamos la letra `E` dentro de la información de formato:

```
double numero = 12300000;
DecimalFormat formato = new DecimalFormat("0.###E0");
campoTexto.setText(formato.format(numero));
```

mediante este código obtenemos lo siguiente:

`1.23E7`

Finalmente, podemos mostrar los números en formato de moneda utilizando el carácter `$` dentro del patrón. Por ejemplo, si usamos el siguiente código:

```
double dinero = 12.34;
DecimalFormat formato = new DecimalFormat("$##.##");
campoTexto.setText(formato.format(dinero));
```

obtendremos lo siguiente:

`$12.34`

en donde el símbolo de moneda (`$` en este caso) se determina de acuerdo con la especificación local. El número siempre se muestra con dos dígitos después del punto decimal (se redondea si es necesario).

La siguiente tabla muestra un resumen de algunos patrones comunes.

| Tipo de datos                   | Valor de ejemplo | Patrón  | Cadena con formato |
|---------------------------------|------------------|---------|--------------------|
| entero                          | 123              | ###     | 123                |
| entero con ceros a la izquierda | 123              | 00000   | 00123              |
| punto flotante                  | 12.34            | ##.##   | 12.34              |
| notación científica             | 12300000         | 0.###E0 | 1.23E7             |
| moneda                          | 12.34            | \$##.## | \$12.34            |

Y la siguiente tabla sintetiza los caracteres de formato que se pueden utilizar para crear patrones:

| Carácter | Significado                                |
|----------|--------------------------------------------|
| #        | inserta un dígito si hay uno               |
| 0        | siempre inserta un dígito                  |
| ,        | inserta una coma                           |
| .        | inserta un punto decimal                   |
| E        | inserta una E seguida de la potencia de 10 |
| \$       | inserta un signo de moneda                 |

La clase `DecimalFormat` está dentro del paquete `java.text.DecimalFormat`, por lo que se necesita la siguiente instrucción `import` en el encabezado de un programa que la utilice:

```
import java.text.DecimalFormat;
```

## PRÁCTICA DE AUTOEVALUACIÓN

- 11.1** Sabemos que el resultado de un cálculo será un número de punto flotante en el rango de 0 a 99, y deben aparecer dos dígitos después del punto decimal. Seleccione un formato apropiado.

### Ejemplo práctico: dinero

Ahora vamos a rastrear el desarrollo de un programa para realizar cálculos con números. En la mayoría de los países el dinero consta de dos partes: dólares y centavos, euros y centavos, libras y peniques. Tenemos una opción: podemos representar un monto de dinero ya sea como cantidad `double` (como 20.25 dólares) o como `int` (2025 centavos). Si utilizamos centavos, tenemos que convertir las cantidades en dólares y centavos, y viceversa. En este caso utilizaremos variables `double` para representar valores.

Vamos a construir un programa que calcula el interés compuesto. Se invierte un monto a una tasa de interés anual específica y se acumula su valor. El usuario introduce la cantidad inicial (como número entero) y una tasa de interés (un número que puede tener un punto decimal) en campos de texto. Después el usuario hace clic en un botón para ver el monto acumulado por año, como se muestra en la figura 11.1.

Al hacer clic en el botón para pasar al siguiente año, el programa debe realizar el siguiente cálculo:

```
montoNuevo = montoAnterior + (montoAnterior * tasa / 100);
```

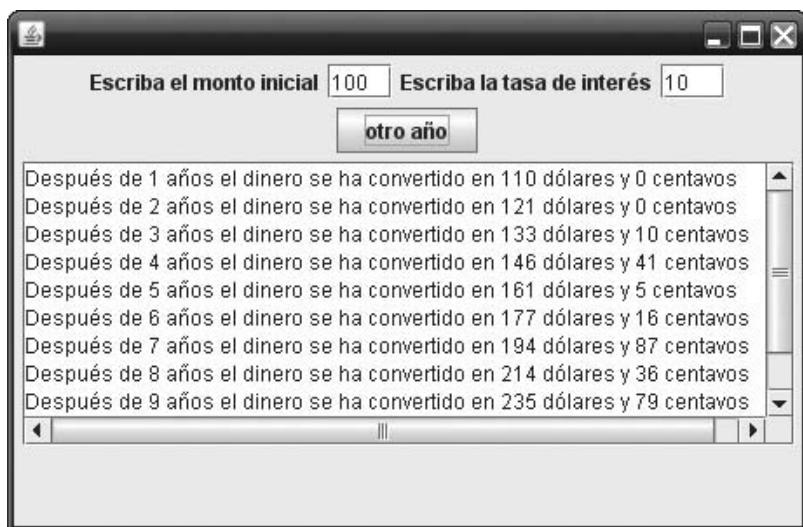


Figura 11.1 Cálculo de intereses.

Al mostrar un monto de dinero, necesitamos que aparezca un número entero de dólares y un número entero de centavos; por ejemplo, si el valor es 127.2341 dólares, necesitamos mostrarlo como 127 dólares y 23 centavos.

Primero veamos la parte de los dólares. Si utilizamos el operador de conversión (`int`) podemos convertir el número `double` en un `int`, truncando la parte fraccionaria:

```
dólares = (int) montoNuevo;
```

Ahora la parte de los centavos. Necesitamos deshacernos de la parte del número correspondiente a los dólares. Podemos restar el número entero de dólares, de manera que un número como 127.2341 se convertiría en 0.2341. Luego multiplicamos ese número por 100.0 para convertirlo en centavos, de manera que 0.2341 se convertiría en 23.41. A continuación usamos `Math.round` para convertirlo al número entero más cercano (23.0). Por último convertimos el valor `double` en un valor `int` mediante (`int`):

```
centavos = (int) Math.round(100 * (montoNuevo - dólares));
```

Ahora podemos mostrar los valores que se convirtieron en forma apropiada. Finalmente tenemos que:

```
monToAnterior = montoNuevo;
```

y esto es de lo que trata la inversión.

A nivel de clase, las declaraciones de instancias son:

```
private int año = 1;
private double monToAnterior;
```

He aquí el programa completo:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Interés extends JFrame implements ActionListener {

 private JLabel etiquetaInicial;
 private JTextField campoInicial;
 private JLabel etiquetaInterés;
 private JTextField campoInterés;
 private JButton botón;
 private JTextArea áreaTexto;

 private int año = 1;
 private double montoAnterior;

 public static void main(String [] args) {
 Interés marco = new Interés();
 marco.setSize(460,300);
 marco.createGUI();
 marco.setVisible(true);
 }

 private void createGUI() {
 setDefaultCloseOperation(EXIT_ON_CLOSE);
 Container ventana = getContentPane();
 ventana.setLayout(new FlowLayout());

 etiquetaInicial = new JLabel("Escriba el monto inicial");
 ventana.add(etiquetaInicial);

 campoInicial = new JTextField(3);
 ventana.add(campoInicial);

 etiquetaInterés = new JLabel("Escriba la tasa de interés");
 ventana.add(etiquetaInterés);

 campoInterés = new JTextField(3);
 ventana.add(campoInterés);

 botón = new JButton("otro año");
 ventana.add(botón);
 botón.addActionListener(this);

 áreaTexto = new JTextArea(10, 40);
 ventana.add(áreaTexto);

 JScrollPane panelDesplazable = new JScrollPane(áreaTexto);
 ventana.add(panelDesplazable);
 }
}
```

```
public void actionPerformed(ActionEvent event) {
 unAño();
}

private void unAño() {
 String nuevaLínea = "\r\n";
 double tasa, montoNuevo;
 int dólares, centavos;

 if (año == 1) {
 montoAnterior = Double.parseDouble(campoInicial.getText());
 }

 tasa = Double.parseDouble(campoInterés.getText());

 montoNuevo = montoAnterior + (montoAnterior * tasa / 100);

 dólares = (int) montoNuevo;
 centavos = (int) Math.round(100 * (montoNuevo - dólares));
 áreaTexto.append("Después de " + Integer.toString(año) + " años "
 + "el dinero se ha convertido en "
 + Integer.toString(dólares) + " dólares y "
 + Integer.toString(centavos) + " centavos" + nuevaLínea);

 montoAnterior = montoNuevo;
 año++;
}
```

Cabe mencionar que agregamos un panel desplazable (`JScrollPane`), para que el área de texto tenga una barra de desplazamiento.

## Ejemplo práctico – iteración

En la programación numérica es muy común escribir iteraciones —ciclos que continúan buscando una solución a una ecuación hasta encontrarla con una precisión adecuada.

Como ejemplo del uso de iteraciones, veamos una fórmula para el seno de un ángulo:

$$\text{sen}(x) = x - x^3/3! + x^5/5! - x^7/7! + \dots$$

(Tenga en cuenta que si necesitamos el seno de un ángulo en un programa no tenemos que usar esta fórmula, ya que está disponible como función de biblioteca).

Podemos ver que cada término se deriva del término anterior con base en la siguiente multiplicación:

$$-x^2/((n+1) \times (n+2))$$

por lo tanto, podemos construir un ciclo que itere hasta que el nuevo término sea menor que cierta cifra aceptable, por decir 0.0001:

```

private double seno(double x) {
 double término, resultado;
 resultado = 0.0;
 término = x;
 for (int n = 1; Math.abs(término) >= 0.0001; n = n + 2) {
 resultado = resultado + término;
 término = - término * x * x / ((n + 1) * (n + 2));
 }
 return resultado;
}

```

en donde el método de biblioteca `abs` calcula el valor absoluto de su parámetro.

## Gráficos

Es común presentar la información matemática, de ingeniería y financiera en forma gráfica. Ahora veremos un programa para trazar funciones matemáticas. Suponga que queremos dibujar la siguiente función:

$$y = ax^3 + bx^2 + cx + d$$

en donde los valores para  $a$ ,  $b$ ,  $c$  y  $d$  se introducen mediante controles deslizables, como en la figura 11.2.

Debemos resolver varias cuestiones de diseño. En primer lugar, queremos ver el gráfico de manera que los valores positivos de la coordenada  $y$  aumenten hacia arriba en la pantalla, mientras que las coordenadas de los píxeles  $y$  se miden hacia abajo. Tendremos que distinguir entre  $x$  y su coordenada de píxel equivalente `píxelX`, y entre  $y$  y `píxelY`.

Ahora tenemos que asegurarnos de que el gráfico se ajuste de manera conveniente al panel; es decir, que no sea demasiado pequeño ni demasiado grande. Al proceso de resolver este problema se le conoce como escalamiento. Vamos a suponer que el área disponible en un panel es de 200 píxeles en la dirección  $x$  y de 200 píxeles en la dirección  $y$ . Diseñaremos el programa de manera que se muestren los valores  $x$  y  $y$  en el rango de -5.0 a +5.0. Por lo tanto, 1 unidad de  $x$  (o de  $y$ ) es de 20 píxeles.

Por último, y ya que utilizaremos el método `drawLine` para dibujar el gráfico, tendremos que dibujar una forma curveada como una gran cantidad de líneas pequeñas. Nos desplazaremos a lo largo de la dirección  $x$ , un píxel a la vez, dibujando una línea desde la coordenada  $y$  equivalente hasta la siguiente. Por cada píxel  $x$ , el programa:

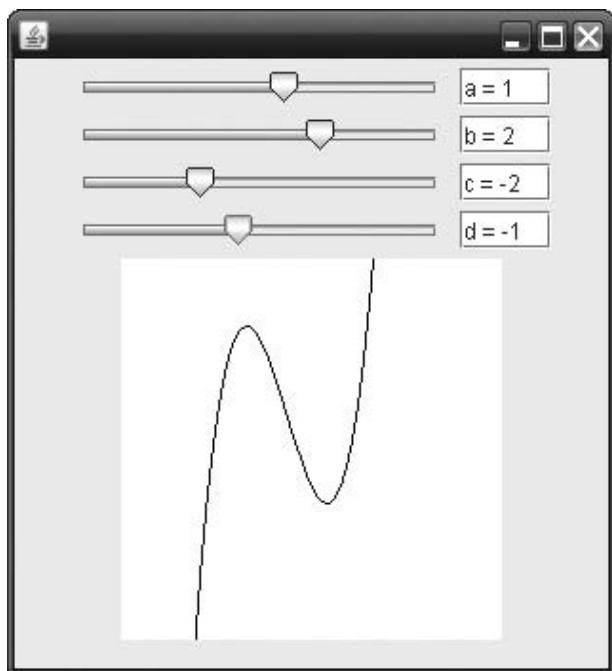
1. Debe calcular el valor de  $x$  a partir del valor del píxel  $x$ .
2. Debe calcular el valor de  $y$ , el valor de la función.
3. Debe calcular el valor del píxel  $y$  a partir del valor de  $y$ .

Para lo cual utilizamos las siguientes instrucciones:

```

x = escalarX(píxelX);
y = laFunción(x);
píxelY = escalarY(y);

```



**Figura 11.2** Programa para dibujar gráficos.

Después el programa avanza al siguiente píxel  $x$ :

```
siguientePíxelX = píxelX + 1;
```

Por último se traza la pequeña sección de la curva:

```
papel.drawLine(píxelX, píxelY, siguientePíxelX, siguientePíxelY);
```

Cabe mencionar que el programa utiliza varios métodos privados para ayudarnos a simplificar la lógica. Además, sólo se utiliza un método para manejar los eventos de los cuatro controles deslizables. He aquí el código completo de este programa para dibujar gráficos.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;

public class Gráfico extends JFrame implements ChangeListener {

 int a, b, c, d;

 private JSlider aDeslizable, bDeslizable, cDeslizable, dDeslizable;
 private JTextField aTexto, bTexto, cTexto, dTexto;
```

```
private JPanel panel;
private int altura = 200, anchura = 200;

public static void main (String[] args) {
 Gráfico frame = new Gráfico();
 frame.setSize(320,350);
 frame.crearGUI();
 frame.setVisible(true);
}

private void crearGUI() {

 setDefaultCloseOperation(EXIT_ON_CLOSE);
 Container ventana = getContentPane();
 ventana.setLayout(new FlowLayout());

 aDeslizante = new JSlider(-5, 5);
 aDeslizante.addChangeListener(this);
 ventana.add(aDeslizante);

 aTexto = new JTextField(4);
 ventana.add(aTexto);

 bDeslizante = new JSlider(-5, 5);
 bDeslizante.addChangeListener(this);
 ventana.add(bDeslizante);

 bTexto = new JTextField(4);
 ventana.add(bTexto);

 cDeslizante = new JSlider(-5, 5);
 cDeslizante.addChangeListener(this);
 ventana.add(cDeslizante);

 cTexto = new JTextField(4);
 ventana.add(cTexto);

 dDeslizante = new JSlider(-5, 5);
 dDeslizante.addChangeListener(this);
 ventana.add(dDeslizante);

 dTexto = new JTextField(4);
 ventana.add(dTexto);

 panel = new JPanel();
 panel.setPreferredSize(new Dimension(anchura, altura));
 panel.setBackground(Color.white);
 ventana.add(panel);
}

public void stateChanged(ChangeEvent e) {
 a = aDeslizante.getValue();
 b = bDeslizante.getValue();
 c = cDeslizante.getValue();
 d = dDeslizante.getValue();
}
```

```

aTexto.setText("a = " + Integer.toString(a));
bTexto.setText("b = " + Integer.toString(b));
cTexto.setText("c = " + Integer.toString(c));
dTexto.setText("d = " + Integer.toString(d));
dibujar();
}

private void dibujar() {
 Graphics papel = panel.getGraphics();
 papel.setColor(Color.white);
 papel.fillRect(0, 0, anchura, altura);
 double x, y, siguienteX, siguienteY;
 int pixelX, pixelY, siguientePixelX, siguientePixelY;
 papel.setColor(Color.black);
 for (pixelX = 0; pixelX <= anchura; pixelX++) {
 x = escalarX(pixelX);
 y = laFunción(x);
 pixelY = escalarY(y);
 siguientePixelX = pixelX + 1;
 siguienteX = escalarX(siguientePixelX);
 siguienteY = laFunción(siguienteX);
 siguientePixelY = escalarY(siguienteY);
 papel.drawLine(pixelX, pixelY, siguientePixelX, siguientePixelY);
 }
}

private double laFunción(double x) {
 return a * x * x * x * x + b * x * x * x + c * x + d;
}

private double escalarX(int pixelX) {
 double xInicial = -5, xFinal = 5;
 double xEscala = anchura / (xFinal - xInicial);
 return (pixelX - (anchura / 2)) / xEscala;
}

private int escalarY(double y) {
 double yInicial = -5, yFinal = 5;
 int coordPixel;
 double yEscala = altura / (yFinal - yInicial);
 coordPixel = (int) (-y * yEscala) +
 (int) (altura / 2);
 return coordPixel;
}
}

```



Si ejecuta este programa, puede alterar los valores de los controles deslizables para ver el efecto de modificar los parámetros. También podemos dibujar ecuaciones cuadráticas (si hacemos el valor del coeficiente  $a$  igual a cero) y líneas rectas.

## ● Excepciones

Si lee este capítulo por primera vez, probablemente sea mejor que omita esta sección ya que trata sobre cosas que no ocurren con mucha frecuencia.

Al escribir un programa que realiza cálculos tenemos que cuidarnos de no exceder el tamaño de los números permitidos. No es como realizar un cálculo en una hoja de papel, en donde los números pueden ser del tamaño que queramos; es algo más parecido a utilizar una calculadora, la cual tiene un límite superior finito en cuanto al tamaño de los números que puede guardar.

Por ejemplo, si declara un `int` de la siguiente manera:

```
int número;
```

debe tener en cuenta que el número más grande que se puede guardar en una variable `int` es extenso, pero se limita a 2147483647. Por lo tanto, si escribe lo siguiente:

```
número = 2147483647;
número = número + 2147483647;
```

el resultado de la suma no se podrá guardar como valor `int`. El programa terminará y aparecerá un mensaje de error. A esto se le conoce como desbordamiento y es una de varias posibles excepciones que pueden surgir al momento en que se ejecuta un programa.

El desbordamiento puede ocurrir con más sutileza que en el ejemplo anterior, especialmente cuando un usuario introduce datos en un cuadro de texto y, por ende, su tamaño es impredecible. Por ejemplo, he aquí un programa simple para calcular el área de una habitación en donde podría ocurrir un desbordamiento:

```
int longitud, área;
longitud = Integer.parseInt(campoTextoEntrada.getText());
área = longitud * longitud;
```

Algunas de las situaciones que podrían provocar un desbordamiento son:

- Sumar dos números grandes.
- Restar un número positivo grande a un número negativo grande.
- Dividir entre un número muy pequeño.
- Multiplicar dos números grandes.

De todo esto podemos concluir que aun cuando un simple cálculo se vea inofensivo, necesitamos vigilarlo. Hay varias formas de lidiar con una excepción:

1. Ignorarla esperando que no vaya a ocurrir, y estar preparados para que el programa falle y/o produzca resultados extraños cuando surja la excepción. Esto está bien para los programadores principiantes, pero no es nada conveniente para los programas reales diseñados para ser robustos.
2. Permitir que surja la excepción pero manejarla escribiendo un manejador de excepciones, como veremos más adelante en el capítulo 16.
3. Evitarla escribiendo comprobaciones para asegurarnos de evitar dicha situación. Por ejemplo, en un programa para calcular el área de una habitación, podemos evitar el desbordamiento si comprobamos el tamaño de los datos de la siguiente manera:

```
if (longitud > 10000) {
 campoTextoRespuesta.setText("valor demasiado grande");
}
```

Ya vimos cómo puede ocurrir un desbordamiento cuando un programa utiliza valores `int`. Podríamos esperar que ocurriera lo mismo si los valores `double` se vuelven demasiado grandes, pero no es así. Si un valor se vuelve demasiado grande, el programa sigue funcionando y el valor toma uno de los valores especiales: `PositiveInfinity` (infinito positivo) o `NegativeInfinity` (infinito negativo) según sea apropiado.

## Fundamentos de programación

- Muchos programas científicos, de ingeniería, matemáticos y estadísticos emplean muchos cálculos. Pero incluso los programas pequeños que tal vez no requieren cálculos utilizan a menudo algo de aritmética.
- El primer paso (y el más importante) es decidir qué tipos de variables utilizar para representar los datos. La principal elección es entre `int` y `double`.
- La biblioteca de funciones matemáticas es invaluable en los programas de este tipo.
- Se puede dar formato a los números para mostrarlos en pantalla.
- Es común utilizar las iteraciones en los cálculos numéricos en los que la solución converge hacia la respuesta. Para ello se requiere un ciclo.
- Las situaciones excepcionales, como el desbordamiento, pueden ocurrir durante los cálculos, y debemos anticiparnos a ellas si queremos que nuestro programa sea robusto en toda circunstancia.

## Errores comunes de programación

- Las situaciones excepcionales, como tratar de dividir entre cero, pueden producir resultados extraños, o el programa puede terminar su ejecución. Cerciórese de que sus programas sean robustos.

## Resumen

- Las principales formas de representar números son mediante `int` o `double`. Estos tipos de datos producen distintas precisiones y rangos.
- Las funciones de biblioteca proveen las funciones matemáticas comunes; por ejemplo, el seno de un ángulo.
- El método `format` de la clase de biblioteca `DecimalFormat` se puede usar para dar formato a los números y mostrarlos en pantalla.
- El programador debe tener en cuenta las excepciones que podrían surgir durante los cálculos.

## Ejercicios

- 11.1 Costo de una llamada telefónica** Una llamada telefónica cuesta 10 centavos por minuto. Escriba un programa que reciba mediante campos de texto la duración de una llamada telefónica, expresada en horas, minutos y segundos, y que muestre el costo de esta llamada en centavos, con una precisión hasta el centavo más cercano.
- 11.2 Conversión de mediciones** Escriba un programa que reciba como entrada una medición utilizando dos campos de texto, expresada en pies y pulgadas. Al hacer clic en un botón hay que convertir la medición a centímetros y mostrarla en un campo de texto, con dos lugares decimales. Hay 12 pulgadas en un pie; una pulgada equivale a 2.54 centímetros.
- 11.3 Clic del ratón** Escriba un programa que muestre un panel. Cuando el usuario haga clic dentro del panel deberá aparecer un panel de opción que muestre la distancia del ratón desde la esquina superior izquierda del panel. Consulte el apéndice A para ver cómo obtener las coordenadas del clic del ratón.
- 11.4 Caja registradora** Escriba un programa que represente una caja registradora. Los montos de dinero se pueden introducir en un campo de texto y se deben sumar al total acumulado cuando se haga clic en un botón. El total acumulado se debe mostrar en otro campo de texto. Debe haber otro botón que permita borrar la suma (hacerla cero).
- 11.5 Suma de enteros** La suma de los enteros de 1 a  $n$  se obtiene mediante la siguiente fórmula

$$\text{suma} = n(n + 1)/2$$

Escriba un programa que reciba como entrada un valor para  $n$  mediante un campo de texto y que calcule la suma de dos formas: primero utilizando la fórmula y después sumando los números mediante un ciclo.

- 11.6 Cálculo de intereses** El programa que se muestra en el texto calcula los intereses año por año. Una alternativa sería utilizar una fórmula para calcular el interés. Si se invierte un monto  $p$  a una tasa de interés  $r$  durante  $n$  años, el valor acumulado  $v$  sería:

$$v = p(1 + r)^n$$

Escriba un programa que acepte valores para  $p$  (en dólares),  $r$  (como porcentaje) y  $n$  mediante campos de texto y que muestre el valor acumulado en otro campo de texto.

- 11.7 Números aleatorios** Los números aleatorios se utilizan con frecuencia en los programas computacionales y de simulación, conocidos como métodos de Monte Carlo. Ya vimos cómo utilizar la clase de biblioteca `Random` que nos permite crear un generador de números aleatorios, como se muestra a continuación:

```
Random aleatorio = new Random();
```

Después llamamos a `nextInt` para obtener un número aleatorio entero en el rango de 0 a 1, como se muestra a continuación:

```
int número = aleatorio.nextInt(2);
```

Escriba un programa para verificar el método generador de números aleatorios. Provea un botón que cree un conjunto de 100 números aleatorios que tengan el valor 0 o 1. Cuente el número de valores iguales a 0 y el número de valores iguales a 1 (deben ser aproximadamente iguales).

- 11.8 Series para e** El valor de  $e^x$  se puede calcular mediante la suma de la siguiente serie:

$$e^x = 1 + x + x^2/2! + x^3/3! + \dots$$

Escriba un programa que reciba como entrada un valor de  $x$  mediante un campo de texto y que calcule  $e^x$  hasta cierto grado de precisión deseado. Compruebe el valor con el valor que se obtiene al usar el método `exp` de la biblioteca `Math`.

- 11.9 Cálculo de impuestos** Escriba un programa que lleve a cabo el cálculo de impuestos. Los impuestos son cero para los primeros \$10,000, pero son del 33% para cualquier monto mayor a ese. Escriba el programa para recibir como entrada un salario en dólares mediante un campo de texto y calcular los impuestos a pagar. Vigile los errores al realizar el cálculo; ¡la respuesta necesita ser precisa hasta el centavo más cercano!

- 11.10 Área de un triángulo** El área de un triángulo cuyos lados son de una longitud  $a$ ,  $b$  y  $c$  es:

$$\text{área} = \sqrt{s(s - a)(s - b)(s - c)}$$

en donde:

$$s = (a + b + c)/2$$

Escriba un programa que reciba como entrada los tres valores de los lados de un triángulo mediante campos de texto y utilice esta fórmula para calcular el área. Su programa debe comprobar primero que las tres longitudes especificadas en realidad formen un triángulo. Por ejemplo,  $a + b$  debe ser mayor que  $c$ .

- 11.11 Raíz cuadrada** La raíz cuadrada de un número se puede calcular mediante iteraciones, como se muestra a continuación. Escriba un programa para hacer esto con un número que se introduzca mediante un campo de texto.

La primera aproximación a la raíz cuadrada de  $x$  es  $x/2$ .

Después las aproximaciones sucesivas se obtienen mediante la siguiente fórmula:

$$\text{siguienteAproximación} = (\text{últimaAproximación}^2 - x)/2 + \text{últimaAproximación}.$$

Compare el valor con el que se obtiene al utilizar el método de biblioteca `sqrt`.

- 11.12 Calculadora matemática** Escriba un programa que actúe como una calculadora matemática. Debe tener botones con los cuales se puedan introducir números, los que se deben mostrar como en la pantalla de una calculadora de escritorio. También debe haber botones para realizar cálculos matemáticos estándar, como seno, coseno, logaritmo natural y raíz cuadrada.

- 11.13 Calculadora de interés** Modifique la parte del cálculo del programa anterior sobre el cálculo de intereses, de manera que se pueda utilizar un número `int` (en vez de `double`) para representar un monto de dinero (expresado en centavos).

- 11.14 Trazador de gráficos** Mejore el programa para dibujar gráficos que vimos en este capítulo de manera que:

- Dibuje los ejes  $x$  y  $y$ .
- Reciba como entrada los coeficientes mediante campos de texto, en vez de controles deslizables (para tener precisión).
- Reciba como entrada un factor de escala horizontal y vertical (acercamiento) mediante controles deslizables.
- Dibuje un segundo gráfico de la misma función, pero con coeficientes distintos.
- Dibuje los gráficos de algunas otras funciones. Una manera de hacer esto sería volver a escribir el método `laFunción`.

**11.15 Integración numérica** Escriba un programa que calcule la integral de una función  $y$  utilizando la “regla del trapecio”. El área bajo el gráfico de la función se divide en  $n$  bandas iguales de longitud  $d$ . Despues el área bajo la curva (la integral) es aproximadamente igual a la suma de todos los (pequeños) trapecios:

$$\text{área} \cong \frac{1}{2}d(y_0 + 2y_1 + 2y_2 + \dots + 2y_{n-1} + y_n)$$

O:

$$\text{área} = (\text{la mitad de la anchura de la banda}) \times (\text{primera} + \text{última} + \text{dos veces la suma de las demás}).$$

Utilice una función para la cual conozca la respuesta y experimente utilizando valores cada vez más pequeños de  $d$ .

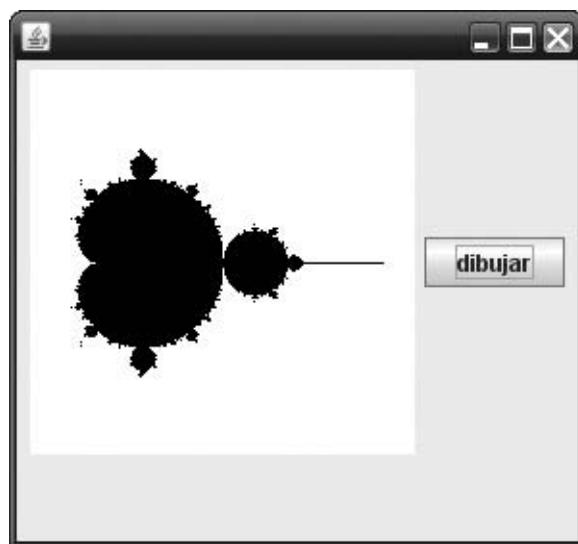
**11.16 Conjunto de Mandelbrot** El conjunto de Mandelbrot (figura 11.3) es una famosa e impactante imagen que se produce al evaluar en forma repetida una fórmula en cada punto, en un espacio de dos dimensiones. Tome un punto con las coordenadas  $x_{\text{inicial}}$  y  $y_{\text{inicial}}$ . Despues calcule en forma repetida nuevos valores de  $x$  y de  $y$  a partir de los valores anteriores, utilizando las siguientes fórmulas:

$$x_{\text{nueva}} = x_{\text{anterior}}^2 - y_{\text{anterior}}^2 - x_{\text{inicial}}$$

$$y_{\text{nueva}} = 2x_{\text{anterior}}y_{\text{anterior}} - y_{\text{inicial}}$$

Los primeros valores de  $x_{\text{anterior}}$  y  $y_{\text{anterior}}$  son  $x_{\text{inicial}}$  y  $y_{\text{inicial}}$ . Para cada iteración, calcule  $r = \sqrt{x_{\text{nueva}}^2 + y_{\text{nueva}}^2}$ . Repita hasta que  $r > 10000$  o hasta llegar a 100 iteraciones, lo que ocurra primero. Si  $r$  es mayor que 10000, coloree de blanco el píxel correspondiente a esta coordenada; en caso contrario coloréelo de negro.

Repita el proceso para todos los puntos con un valor de  $x$  entre  $-1.0$  y  $+2.0$ , y con un valor de  $y$  en el rango de  $-2.0$  a  $+2.0$ .



**Figura 11.3** El conjunto de Mandelbrot.

A medida que avanza la iteración, partiendo de valores específicos de  $x_{inicial}$  y  $y_{inicial}$ , el valor de  $r$  algunas veces permanece razonablemente pequeño (cerca de 1.0). Para otros valores de  $x_{inicial}$  y  $y_{inicial}$  el valor de  $r$  se vuelve rápidamente muy grande y tiende a dispararse hacia el infinito.

A propósito, algunas veces podemos ver imágenes del conjunto de Mandelbrot que son la imagen espejo de la que se muestra en la figura 11.3. Esas imágenes se generan mediante las siguientes fórmulas, que son ligeramente distintas:

$$x_{nueva} = x_{anterior}^2 - y_{anterior}^2 + x_{inicial}$$

$$y_{nueva} = 2x_{anterior}y_{anterior} + y_{inicial}$$

## Respuesta a la práctica de autoevaluación

```
11.1 double respuesta;
 DecimalFormat formato = new DecimalFormat("##.##");
 campoTexto.setText(formato.format(respuesta));
```

# CAPÍTULO 12



## Objetos de tipo **ArrayLists**

En este capítulo conoceremos:

- El concepto de objeto **ArrayList**.
- Cómo usar tipos genéricos.
- Cómo usar la instrucción **for** mejorada.
- Cómo mostrar un objeto **ArrayList**.
- Cómo agregar y eliminar elementos de un objeto **ArrayList**.
- Cómo obtener el tamaño de un objeto **ArrayList**.
- El concepto de índice.
- Cómo llevar a cabo operaciones comunes en un objeto **ArrayList** por ejemplo, realizar búsquedas rápidas (*lookup*), operaciones aritméticas y búsquedas detalladas (*search*);

### ● Introducción

Un objeto **ArrayList** es una colección de datos tal como una lista de compras, una lista de nombres de personas o un conjunto de cifras de precipitación pluvial. Java provee herramientas para crear un nuevo objeto **ArrayList**, y para agregar y eliminar elementos de estos objetos. Al igual que otras variables, un objeto **ArrayList** se mantiene en la memoria principal (RAM) y, por lo tanto, es invisible, a menos que mostremos la información mediante un área de texto, por ejemplo.

Un objeto **ArrayList** tiene un nombre que lo identifica como un todo. Pero también podemos hacer referencia a los elementos individuales dentro de un objeto **ArrayList** con base en su posición. A esta posición se le conoce como índice. Los valores de los índices son enteros y empiezan en 0.

Utilizaremos como ejemplo una lista de compras, a la cual le agregaremos un elemento a la vez. Después de agregar varios elementos a la lista, la información que contiene se puede mostrar en pantalla, como en la figura 12.1.



**Figura 12.1** Cómo mostrar un objeto `ArrayList` en un área de texto.

Los objetos `ArrayList` constituyen una buena introducción al uso de las estructuras de datos, ya que son convenientes de usar. En este capítulo exploraremos el uso de los objetos `ArrayList` como estructuras de datos; puede leerlo y estudiarlo de manera independiente a los capítulos sobre matrices.

## ● Creación de un objeto `ArrayList` y los tipos genéricos

La clase `ArrayList` se incluye dentro del paquete `util` de Java y por lo tanto se requiere la siguiente instrucción `import` en el encabezado de cualquier programa que utilice un objeto `ArrayList`:

```
import java.util.*;
```

Podemos crear un objeto `ArrayList` en forma similar a cualquier otro objeto. Hay que asignarle un nombre conveniente y usar la palabra clave `new`:

```
ArrayList <String> lista = new ArrayList <String> ();
```

Esto crea un objeto `ArrayList` vacío. En breve veremos cómo agregar elementos a una lista.

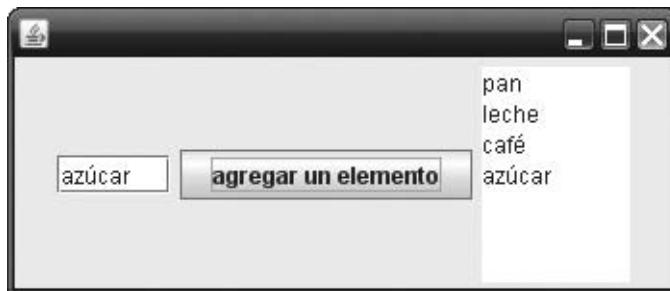
La notación `<>` encierra el nombre de una clase. Ésta es la clase de los objetos que contendrá el objeto `ArrayList`. A esto se le conoce como característica *genérica* de Java. Esta clase puede ser cualquier clase de la biblioteca de Java (por ejemplo: `String`, `Button`, `IconImage`) o cualquier clase que el programador escriba. Una vez que haya declarado un objeto `ArrayList`, sólo debe agregarle elementos que sean instancias de la clase especificada (u objetos de una subclase).

Un objeto `ArrayList` sólo contiene objetos, por lo que no le podemos agregar entidades declaradas como `int`, `double` o `boolean`. Para solucionar este problema podemos guardar estos valores como cadenas de texto; más adelante en el capítulo veremos cómo hacerlo.

## ● Cómo agregar elementos a una lista

Una manera de colocar elementos en un objeto `ArrayList` es utilizando el método de biblioteca `add`. Por ejemplo:

```
lista.add("huevos");
```



**Figura 12.2** Cómo agregar elementos a una lista de compras.

En este ejemplo, el nombre del objeto `ArrayList` es `lista`. El método `add` agrega el elemento al final del objeto `ArrayList` existente. Su parámetro es el valor que se agregará al objeto `ArrayList`; en este caso, la cadena “huevos”.

Con frecuencia, la información que agregamos a un objeto `ArrayList` es un valor que se obtiene del usuario. El programa de ejemplo que se muestra en la figura 12.2 permite al usuario agregar elementos a un objeto `ArrayList`.

Este programa responde al clic de un botón y llama al método `add` para colocar un elemento de la lista de compras al final del objeto `ArrayList`:

```
private void agregarUnElemento(ArrayList <String> lista) {
 lista.add(campoTexto.getText());
}
```

Observe que en el encabezado del método incluimos el tipo del objeto `ArrayList`: `<String>` en este ejemplo. Para llamar a este método podemos usar la siguiente instrucción:

```
agregarUnElemento(miLista);
```

sin la descripción del tipo del objeto `ArrayList`.

Un objeto `ArrayList` se expande según sea necesario para alojar todos los datos que se le agreguen. Es como si estuviera hecho de elástico.

## ● La longitud de una lista

Para averiguar qué tan largo es un objeto `ArrayList` podemos usar el método de biblioteca `size`. Por ejemplo:

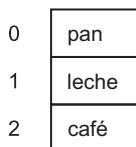
```
int numeroDeElementos = lista.size();
```

He aquí un método que muestra un panel de opción que contiene el número de elementos actuales en el objeto `ArrayList`:

```
private void mostrarLongitud(ArrayList <String> lista) {
 JOptionPane.showMessageDialog(null, Integer.toString(lista.size()));
}
```

## Índices

Para hacer referencia a los elementos individuales en un objeto **ArrayList**, el programa utiliza un *índice*. Éste es un entero que indica a cuál elemento se hace referencia. El primer elemento tiene un valor de índice 0, el segundo de 1, etc. Podemos visualizar el objeto **ArrayList** de compras como una tabla, como se muestra en la figura 12.3, con los valores de los índices a un lado.



**Figura 12.3** Diagrama de un objeto **ArrayList** que muestra los índices.

## Cómo mostrar un objeto **ArrayList**

Un objeto **ArrayList** está contenido en la memoria principal (RAM), por lo que generalmente es invisible. Ahora veremos un método que muestra el contenido de un objeto **ArrayList** (la lista de compras llamada **lista**) en un área de texto. Este método produce la pantalla que se muestra en la figura 12.4.



**Figura 12.4** Cómo mostrar un objeto **ArrayList**.

El programa utiliza el método **size** que nos indica qué tan larga es la lista. Usamos una instrucción **for** pues sabemos que se necesita una repetición. Utilizamos el método **get** para obtener los valores del objeto **ArrayList**. El parámetro para **get** especifica el valor del índice del elemento requerido. **get** simplemente obtiene el valor (hace una copia del mismo) sin perturbar la lista.

```
private void mostrar(ArrayList <String> lista) {
 final String nuevaLínea = "\n";
 áreaTexto.setText("");
 for (int índice = 0; índice < lista.size(); índice++) {
 áreaTexto.append(lista.get(índice) + nuevaLínea);
 }
}
```



**Figura 12.5** Barras de desplazamiento.

Algunas veces un área de texto es demasiado pequeña como para mostrar todo el contenido de un objeto **ArrayList**. Para resolver este problema podemos adjuntar barras de desplazamiento (horizontal y vertical) al área de texto; en la figura 12.5 se muestra un ejemplo. Estas barras aparecen sólo si es necesario (cuando los datos son demasiado grandes como para poder mostrarlos en su totalidad). El código adicional para proveer barras de desplazamiento es el siguiente:

```
JScrollPane panelDesplazable = new JScrollPane(áreaTexto);
ventana.add(panelDesplazable);
```

en donde **áreaTexto** es el nombre del área de texto.

## ● La instrucción for mejorada

Es muy común utilizar instrucciones **for** en conjunto con los objetos **ArrayList**. En esas ocasiones en que el programa necesita procesar cada uno de los elementos en un objeto **ArrayList**, hay una excelente forma de hacerlo mediante la instrucción **for** mejorada. Podemos replantear el método anterior para mostrar un objeto **ArrayList**, como se muestra a continuación:

```
private void mostrar(ArrayList <String> lista) {
 final String nuevaLínea = "\n";
 áreaTexto.setText("");
 for (String elemento : lista) {
 áreaTexto.append(elemento + nuevaLínea);
 }
}
```

Este método es más eficiente y corto. Podemos leer la instrucción **for** como “para todas las cadenas **elemento** en la lista”. El carácter de dos puntos se puede leer como “en”. El ciclo se repite para todos los elementos de la lista. En cada repetición, la variable **elemento** contiene el valor del elemento en la lista.

La clase que se declara como parte de la instrucción **for** (**String** en este ejemplo) debe coincidir con la clase que contiene el objeto **ArrayList**. La variable (**elemento** en este ejemplo) puede tener cualquier nombre, al igual que cualquier otra variable.

Hay dos desventajas en cuanto a usar la instrucción `for` mejorada. En primer lugar, sólo podemos usarla cuando necesitamos procesar **todos** los elementos en un objeto `ArrayList`. En segundo lugar, el valor del índice no está disponible dentro del ciclo. En la siguiente sección veremos un programa de ejemplo.

## Cómo utilizar valores de índice

Ya hemos visto cómo mostrar un objeto `ArrayList`. También podemos mostrar los valores a un lado de sus valores de índice, como en la figura 12.6. El código es:

```
private void mostrarConÍndices(ArrayList <String> lista) {
 final String nuevaLínea = "\n";
 final String tab = "\t";
 áreaTexto.setTabSize(3);
 áreaTexto.setText("");
 for (int índice = 0; índice < lista.size(); índice++) {
 áreaTexto.append(Integer.toString(índice)
 + tab
 + lista.get(índice) + nuevaLínea);
 }
}
```

La figura 12.7 muestra un programa que permite al usuario desplegar el valor en un índice específico. El código para mostrar el elemento es:

```
private void mostrarElemento(ArrayList <String> lista) {
 int índice;
 índice = Integer.parseInt(campoTextoIndice.getText());
 valor.setText(lista.get(indice));
}
```



**Figura 12.6** Cómo mostrar un objeto `ArrayList` junto con los valores de los índices.



Figura 12.7 Cómo mostrar un elemento de un objeto `ArrayList`.

Este programa obtiene el valor de un índice de un campo de texto y lo convierte de su representación de cadena en un `int` mediante `Integer.parseInt`; después lo coloca en la variable `índice`. A continuación se utiliza el valor del índice para acceder al elemento correspondiente en la lista. Se utiliza el método `get` para obtener el valor del objeto `ArrayList`.

## PRÁCTICA DE AUTOEVALUACIÓN

**12.1** En la figura 12.7, ¿qué elemento está en el índice con el valor de 1?

### Cómo eliminar elementos de un objeto `ArrayList`

Ya hemos visto cómo agregar elementos a un objeto `ArrayList`. Ahora veremos cómo eliminar información. El método `remove` de la clase `ArrayList` elimina el elemento en un valor de índice específico. Por lo tanto, si tenemos un objeto `ArrayList` llamado `lista`, podemos eliminar el elemento en el valor de índice 3 mediante la siguiente instrucción:

```
lista.remove(3);
```

Cuando esto ocurre, se cierra el vacío creado, recorriendo los elementos restantes hacia arriba. El objeto `ArrayList` se encoge al tamaño necesario.

Podemos vaciar por completo un objeto `ArrayList` utilizando el método `clear`, como en la siguiente instrucción:

```
lista.clear();
```

## PRÁCTICA DE AUTOEVALUACIÓN

**12.2** ¿Cuál es el tamaño de un objeto `ArrayList` después de usar el método `clear` en él?

## Cómo insertar elementos dentro de un objeto ArrayList

Hemos visto cómo agregar elementos al final de una lista con el método `add`. También es fácil insertar elementos dentro del cuerpo de una lista, y se hace con el mismo método `add`. Dada una lista existente, podemos hacer lo siguiente:

```
lista.add(5, "té");
```

El elemento que antes se encontraba en el valor de índice 5 se desplaza hacia abajo en la lista, junto con todos los elementos subsiguientes. El objeto `ArrayList` se expande para dar cabida al nuevo elemento.

## Búsquedas rápidas (lookup)

Podemos usar una tabla tal como un objeto `ArrayList` de manera conveniente para búsquedas rápidas. Por ejemplo, podemos construir un objeto `ArrayList` (como se muestra en la figura 12.8) que contenga los nombres de los meses de enero a diciembre. Si alguien nos proporciona un mes expresado como número (del 1 al 12), podemos utilizar la tabla para convertir el número en el texto equivalente.

|   |         |
|---|---------|
| 0 | Enero   |
| 1 | Febrero |
| 2 | Marzo   |
| 3 | etc.    |

Figura 12.8 Diagrama de un objeto `ArrayList` para convertir enteros en nombres de meses.

La figura 12.9 muestra la apariencia que tiene el programa para el usuario. Creamos un objeto `ArrayList`:

```
ArrayList <String> meses = new ArrayList <String> ();
```

y después colocamos los valores de cadena en él:

```
meses.add("Enero");
meses.add("Febrero");
meses.add("Marzo");
etc.
```



Figura 12.9 El programa para convertir los meses.

El usuario introduce un número en un campo de texto y hace clic en el botón. El programa responde al evento de la siguiente manera:

```
public void actionPerformed(ActionEvent event) {
 int númeroMes;
 String nombreMes;

 númeroMes = Integer.parseInt(campoTextoNúmeroMes.getText());
 nombreMes = meses.get(númeroMes - 1);
 campoTextoNombreMes.setText(nombreMes);
}
```

Los números que representan los meses van del 1 al 12, mientras que los valores de los índices empiezan desde 0. Por lo tanto, debemos restar 1 al número del mes, como se muestra en el código anterior, para convertirlo en un índice apropiado. Después utilizamos el método **get** para obtener el nombre del mes.

Utilizar una tabla de búsqueda rápida como la anterior es una alternativa a escribir una serie de instrucciones **if** para llevar a cabo la conversión. Las instrucciones **if** equivalentes empezarían así:

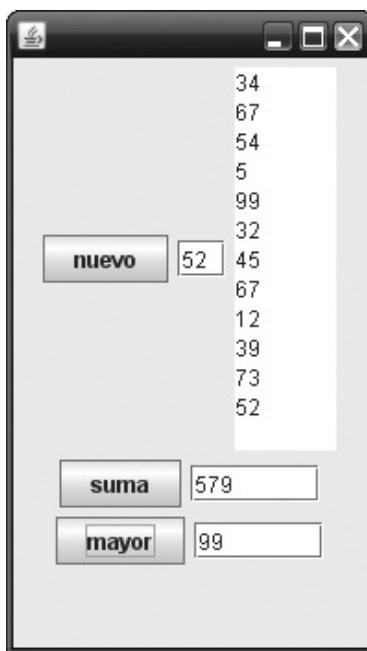
```
if (númeroMes == 1) {
 nombreMes = "Enero";
}
else {
 if (númeroMes == 2) {
 nombreMes = "Febrero";
 }
}
```

Otra opción sería usar una instrucción **switch**. Al emplear instrucciones **if** o una instrucción **switch** se usan acciones para llevar a cabo la conversión. En contraste, al usar una tabla (como un objeto **ArrayList**) se expresa la información de conversión de una manera más eficiente dentro de la tabla.

## ● Operaciones aritméticas en un objeto **ArrayList**

Ahora veremos un objeto **ArrayList** llamado **números** que contiene números enteros, y vamos a realizar operaciones aritméticas con esos números. La figura 12.10 muestra un programa que permite al usuario introducir números en un objeto **ArrayList**. Después un botón hace que aparezca la suma de los números y otro botón hace que aparezca el número más grande.

He aquí un método para sumar todos los números en una lista. No podemos establecer un objeto **ArrayList** para guardar valores **int** debido a que sólo puede contener los objetos apropiados. En vez de ello vamos a establecer un objeto **ArrayList** para que contenga representaciones de cadena de



**Figura 12.10** Operaciones aritméticas en un objeto **ArrayList**.

los números. Cada cadena se convierte en un valor **int** mediante **Integer.parseInt**. Por último, se suma el entero a un total acumulado llamado **suma**, el cual en un principio es igual a 0. Después del ciclo se coloca el valor de **suma** en un campo de texto. Como vamos a procesar todos los números del objeto **ArrayList**, vamos a utilizar una instrucción **for** mejorada para ejecutar un ciclo.

```
private void obtenerSuma(ArrayList <String> números) {
 int suma = 0;
 for (String cadena : números) {
 int número = Integer.parseInt(cadena);
 suma = suma + número;
 }
 campoSuma.setText(Integer.toString(suma));
}
```

A continuación estudiaremos un método para encontrar el elemento más grande en una lista de números. Utilizaremos una variable llamada **mayor** para llevar la cuenta del valor más grande. Al principio, **mayor** es igual al valor en el índice 0 del objeto **ArrayList** (el primer elemento de la lista). Este valor se copia de la lista, pero antes de poder utilizarlo debemos convertirlo en **int** mediante **Integer.parseInt**.

Utilizamos una instrucción `for` mejorada para procesar los números de la lista. Cada elemento de la lista se compara con `mayor` y, si es mayor, se actualiza el valor de `mayor`. Así que cuando llegamos al final de la lista, `mayor` contiene el valor más grande.

```
private void obtenerMayor(ArrayList <String> números) {
 int mayor;

 mayor = Integer.parseInt(números.get(0));
 for (String cadena : números) {
 int número = Integer.parseInt(cadena);
 if (número > mayor) {
 mayor = número;
 }
 }
 campoMayor.setText(Integer.toString(mayor));
}
```

## PRÁCTICA DE AUTOEVALUACIÓN

- 12.3** Modifique este método de una manera simple para que encuentre el elemento más pequeño de la lista.

Estas dos secciones del programa ilustran una característica común de los programas que manipulan listas: cada vez que debamos procesar todos los elementos de una lista, probablemente sea más apropiado utilizar una instrucción `for` en vez de una instrucción `while`.

## Búsquedas detalladas (search)

Nuestro siguiente programa lleva a cabo una búsqueda detallada. Vamos a suponer que ya hay una lista (por ejemplo, la lista de compras) y queremos buscar cierto elemento en ella. El usuario introduce el elemento deseado (por ejemplo, té) en un campo de texto y hace clic en un botón, como se muestra en la figura 12.11.

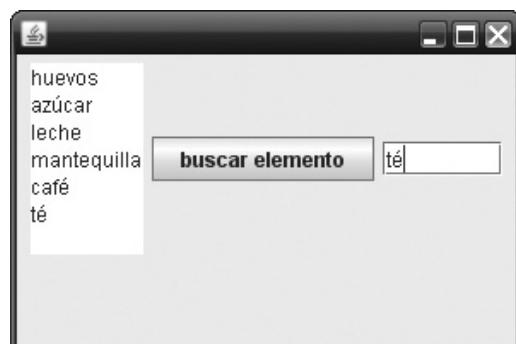


Figura 12.11 Búsqueda detallada en un objeto `ArrayList`.

El programa empieza desde el primer elemento de la lista y avanza por la misma en forma descendente, un elemento a la vez, tratando de encontrar el elemento deseado. Si no lo encuentra, el valor del índice se vuelve igual a la longitud de la lista (**size**) y el ciclo termina. Si encuentra el elemento, la variable **boolean** llamada **encontró** se vuelve **true** y el ciclo termina.

```
private void buscar(ArrayList <String> lista) {
 int longitud;
 int índice;
 boolean encontró;
 String elementoBuscado;

 longitud = lista.size();
 elementoBuscado = campoTexto.getText();
 encontró = false;
 índice = 0;
 while ((encontró == false) && (índice < longitud)) {
 ((lista.get(índice)).equals(elementoBuscado)) {
 encontró = true;
 JOptionPane.showMessageDialog(null, "Se encontró el elemento");
 }
 else {
 índice++;
 }
 }
}
```

Éste es un método clásico de búsqueda en serie. Utiliza un ciclo **while** en vez de un ciclo **for**, porque no necesariamente tenemos que procesar todos los elementos en la lista.

## Fundamentos de programación

Los objetos **ArrayList** son tal vez el tipo más simple de estructura de datos que proporciona Java. Permiten ensamblar, mostrar y manipular una lista de objetos. Una estructura de datos es un grupo de elementos de datos que se pueden procesar de manera uniforme. Una estructura de datos tal como un objeto **ArrayList** se mantiene en la memoria principal de la computadora (no en el almacenamiento secundario), de manera que sólo existe mientras se ejecuta el programa. Cuando éste termina, la estructura de datos se destruye.

Las listas son una de las estructuras clásicas en computación. Uno de los lenguajes más antiguos y venerados, conocido como LISP (abreviación de Procesamiento de LISTAS en inglés), solamente utiliza listas. Una lista es una secuencia de elementos que puede aumentar y reducir su tamaño. Podemos agregar elementos al final de una lista y eliminarlos de cualquier parte de la misma. Además se pueden modificar los valores de los elementos dentro de una lista. Por ende, una lista es una estructura flexible para representar una colección de elementos relacionados de cierta manera.





### Fundamentos de programación (continúa)

Otro de los tipos principales de estructuras de datos es la matriz (que veremos en el capítulo 13). Una matriz es una colección de elementos de datos similares, cada uno de los cuales se distingue mediante un índice. Se pueden insertar y eliminar elementos del interior del cuerpo de un objeto **ArrayList**, pero las matrices no soportan esta capacidad. Un objeto **ArrayList** se expande y contrae para dar cabida a los datos requeridos, pero una matriz siempre tendrá un tamaño fijo. No necesitamos usar ninguna sintaxis especial para acceder a un objeto **ArrayList**, mientras que las matrices requieren el uso de corchetes.

El método para encontrar el elemento mayor en una lista de números es un problema clásico en la programación, al igual que el método de búsqueda detallada.

## Errores comunes de programación

Un error común es pensar que los valores de los índices empiezan en 1 (ya que empiezan en 0).

## Nuevos elementos del lenguaje

La notación `<>` describe la clase de objetos que contendrá un objeto **ArrayList**.

La instrucción **for** mejorada tiene la siguiente estructura:

```
for (String s : lista) {
 // cuerpo del ciclo
}
```

## Resumen

Un objeto **ArrayList** es un ejemplo de estructura de datos. Un objeto **ArrayList** puede dar cabida a cualquier cantidad de objetos, ya que crece y se encoge según sea necesario para alojar los elementos a medida que éstos se agregan y eliminan. Al crear un objeto **ArrayList** describimos lo que va a contener mediante la notación `<>`.

Un objeto **ArrayList** recibe un nombre que lo identifica como un todo. Los elementos individuales dentro de un objeto **ArrayList** se identifican mediante un valor de índice único: un entero. Los valores de los índices empiezan en 0 y llegan hasta el tamaño necesario para identificar a todos los elementos del objeto **ArrayList**. Los valores de los índices no se almacenan en el objeto **ArrayList**.

Un programa puede agregar elementos al final de un objeto **ArrayList**, eliminar un elemento, modificarlo o insertarlo en cualquier parte dentro del objeto **ArrayList**. Los métodos disponibles son:

- **add** – agrega un elemento al final o en la parte media de un objeto **ArrayList**.
- **get** – obtiene un elemento.
- **remove** – elimina un elemento.
- **set** – reemplaza un elemento.
- **clear** – elimina todos los elementos.
- **size** – devuelve la longitud de la lista.

## Ejercicios

- 12.1** Escriba un programa en el que se elimine un elemento de un objeto **ArrayList**. Incluya un botón “eliminar” para eliminar el elemento que se especifique como un índice en un campo de texto.
- 12.2** Agregue un botón al programa del ejercicio 12.1 para hacer que se vacíe el objeto **ArrayList**. Use el método **clear**.
- 12.3** Modifique el programa del ejercicio 12.1 de manera que un elemento del objeto **ArrayList** se pueda reemplazar por algún otro texto. Por ejemplo, “leche” se reemplaza por “azúcar”. Incluya un botón identificado como “reemplazar” para llevar a cabo esta acción. La posición (valor de índice) del elemento a reemplazar se introduce en un campo de texto. El nuevo texto también se introduce en un campo de texto.
- 12.4** Escriba un programa que permita insertar elementos o eliminarlos de cualquier posición dentro de un objeto **ArrayList**, mediante los botones apropiados.
- 12.5** **Búsqueda** Mejore el método de búsqueda, de manera que muestre un mensaje para indicar si el elemento requerido se encuentra o no en el objeto **ArrayList**.
- 12.6** **Cola** Use un objeto **ArrayList** para implementar una cola (o fila). Una fila se forma cuando hacemos cola en la caja de un supermercado o cafetería de autoservicio. Implemente lo siguiente:
- Colocar el nombre de una persona en la parte final de la cola mediante un botón y un campo de texto.
  - Eliminar un nombre de la cabeza de la cola mediante un botón.
  - Mostrar toda la cola en un área de texto.

Nota: La cabeza de la cola es el primer elemento de la lista; el elemento con el índice 0.

- 12.7** **Pila** Implemente una pila mediante un objeto **ArrayList**. En algunos juegos de cartas como el solitario, las cartas se reparten boca abajo en una pila. Después las recogemos de la parte superior de ésta. Así que agregamos y eliminamos cartas del mismo extremo: la parte superior. Implemente lo siguiente:
- Colocar una carta en la pila mediante un botón y un campo de texto.
  - Eliminar una carta de la pila mediante un botón.
  - Mostrar la pila en un área de texto.

Nota: La parte superior de la pila es el primer elemento en el objeto **ArrayList**; el elemento con el índice 0.

## Respuestas a las prácticas de autoevaluación

- 12.1** leche.
- 12.2** El tamaño se vuelve cero.
- 12.3** Cambie el signo mayor que por un signo menor que.

# CAPÍTULO 13



## Arreglos

En este capítulo conoceremos cómo:

- Declarar un arreglo.
- Utilizar un índice.
- Obtener el tamaño de un arreglo.
- Pasar arreglos como parámetros.
- Usar la instrucción **for** mejorada.
- Inicializar un arreglo.
- Llevar a cabo operaciones típicas tales como las búsquedas rápidas (lookup) y detalladas (search).
- Crear arreglos de objetos.

### ● Introducción

Hasta ahora en este libro hemos descrito, con excepción de los objetos **ArrayList**, elementos de datos (variables) individuales y aislados. Por ejemplo:

```
int conteo, suma;
String nombre;
```

Estos elementos de datos viven por su propia cuenta, desempeñando funciones útiles en los programas como contadores, sumas o cualquier otra cosa. Podemos considerar estas variables como lugares en la memoria que tienen nombres individuales.

Por otra parte, en la vida lidiamos muy a menudo con datos que no están aislados, sino agrupados en una colección de información. Algunas veces la información se encuentra en tablas. Algunos ejemplos son el itinerario de un tren, un directorio telefónico o el estado de cuenta de un banco. En la programación, a estas cosas se les conoce como estructuras de datos. La información de una

tabla está interrelacionada de alguna manera. El arreglo es uno de los tipos más simples de estructura de datos en la programación. Un arreglo se puede considerar simplemente como una tabla, con una sola fila de información (también podemos visualizar una tabla como una sola columna de información). Esta podría ser una tabla de números, de cadenas de texto o de cualquier otra cosa.

En este capítulo veremos arreglos de números, de cadenas de texto y de otros objetos, como objetos gráficos.

La figura 13.1 muestra un arreglo de números.

|    |    |    |    |   |    |
|----|----|----|----|---|----|
| 23 | 54 | 96 | 13 | 7 | 32 |
|----|----|----|----|---|----|

**Figura 13.1** Un arreglo de números.

Este arreglo podría representar las edades de un grupo de personas en una fiesta. La figura 13.2 muestra una tabla de palabras que guarda los nombres de los miembros de un grupo musical:

|      |      |        |       |
|------|------|--------|-------|
| John | Paul | George | Ringo |
|------|------|--------|-------|

**Figura 13.2** Un arreglo de cadenas de texto.

En Java, a una tabla como ésta se le denomina arreglo. En la programación, a un componente del arreglo se le conoce como *elemento* y nos referimos a éste por su posición en el arreglo, la cual se conoce como *índice* (en el mundo de la programación, algunas veces se utiliza el término *componente* en vez de elemento, y el término *subíndice* en vez de índice). Para nosotros los humanos, el nombre **John** está en la primera posición de esta tabla, pero en Java a la primera posición en un arreglo se le denomina la posición cero. Las posiciones en un arreglo son cero, primera, segunda, tercera, etc. Por ende, la cadena de texto **Ringo** se encuentra en la tercera posición del arreglo anterior. En consecuencia, podemos visualizar un arreglo, junto con sus índices, como se muestra en la figura 13.3:

|          |      |      |        |       |
|----------|------|------|--------|-------|
| Arreglo: | John | Paul | George | Ringo |
| Índices: | 0    | 1    | 2      | 3     |

**Figura 13.3** Un arreglo de cadenas y sus índices.

Recuerde que los índices no se guardan en la memoria de la computadora, únicamente los datos. Los índices son la forma en que podemos localizar la información en un arreglo.

En la figura 13.4 se muestra otro arreglo que contiene números. También se muestran los índices para el arreglo.

|          |    |    |    |    |   |    |
|----------|----|----|----|----|---|----|
| Arreglo: | 23 | 54 | 96 | 13 | 7 | 32 |
| Índices: | 0  | 1  | 2  | 3  | 4 | 5  |

**Figura 13.4** Un arreglo de números con sus índices.

En un programa (al igual que en la vida real) por lo general tenemos que realizar las siguientes operaciones con los arreglos:

- Crear el arreglo: indicar cuál es su longitud y qué tipos de cosas va a guardar.
- Colocar valores en el arreglo (por ejemplo, introducir números en un directorio telefónico personal).
- Mostrar el contenido del arreglo en la pantalla (un arreglo se guarda en la memoria de la computadora y, por ende, es invisible).
- Buscar en el arreglo cierto valor (por ejemplo, buscar en el itinerario de trenes para encontrar un tren a una hora conveniente).
- Sumar el contenido del arreglo (por ejemplo, averiguar cuánto gastó un cliente en el supermercado).

En este capítulo veremos cómo llevar a cabo estas acciones una a la vez, para después juntarlas en un programa completo. Empezaremos por analizar los arreglos de números. Nuestro plan a lo largo de este capítulo es desarrollar un programa con la distribución en pantalla de la figura 13.5 en la página 252. El programa usa un arreglo que guarda los datos sobre la precipitación pluvial para los siete días de la semana (de lunes a domingo). El usuario del programa puede modificar el valor de cualquier elemento de datos individual en el arreglo. Se debe mostrar el mayor de los números en el arreglo.

## Cómo crear un arreglo

En Java un arreglo se declara justo igual que cualquier otro objeto mediante la palabra clave `new`, por lo general en la parte superior de una clase o de un método. El programador debe dar al arreglo un nombre como se muestra a continuación:

```
int[] edades = new int[6];
String[] grupo = new String[4];
```

La variable `edades` está ahora lista para guardar un arreglo de enteros. Al igual que con cualquier otra variable, es común (y una muy buena idea) elegir un nombre para el arreglo que describa con claridad su función. El nombre servirá para el arreglo completo —la colección de datos completa. Las reglas para elegir el nombre de un arreglo son iguales que para cualquier otro nombre de variable en Java. El número entre corchetes después del nombre del arreglo representa su tamaño.

El arreglo `edades` es lo bastante grande como para contener seis números, en donde sus índices van de 0 a 5. El arreglo llamado `grupo` es lo bastante grande como para contener cuatro cadenas de texto. Los índices van de 0 a 3.

### PRÁCTICA DE AUTOEVALUACIÓN

- 13.1** Declare un arreglo para guardar los datos de la precipitación pluvial para cada uno de los siete días de la semana.

## Índices

Para hacer referencia a un elemento individual de un arreglo, un programa especifica el valor de un índice (algunas veces llamado subíndice). Por lo tanto, en el ejemplo anterior `edades[3]` hace referencia al elemento en el arreglo con el índice 3: el valor 13 en este caso. De manera similar, `grupo[2]` contiene la cadena `George`. Recuerde que los índices empiezan en 0, por lo que un arreglo de longitud 4 tiene índices que van de 0 a 3. En consecuencia, la referencia a `grupo[4]` es un error. El programa se detendrá y aparecerá un mensaje de error.

En resumen, los valores de los índices:

- Empiezan en cero.
- Son enteros.
- Llegan hasta un número menor que el tamaño del arreglo (el valor que se especifica al momento de declarar el arreglo).

Algunas veces, como veremos más adelante, es conveniente utilizar el valor de una variable como índice. En tales casos utilizamos variables `int` como índices.

Podemos recibir como entrada el valor de un elemento de un arreglo mediante un campo de texto:

```
edades[2] = Integer.parseInt(campoTexto.getText());
grupo[3] = campoTexto.getText();
```

y de manera similar podemos mostrar los valores en pantalla:

```
campoTexto.setText("la primera edad es " + Integer.toString(edades[0]));
campoTexto.setText("el 4o. miembro del grupo es " + grupo[3]);
```

Este último ejemplo muestra lo cuidadosos que debemos ser con los índices de los arreglos.

Podemos cambiar los valores de elementos individuales de un arreglo mediante instrucciones de asignación, como en el siguiente ejemplo:

```
edades[3] = 99;
grupo[2] = "Mike";
```

En todos estos fragmentos de programa, hacemos referencia a los elementos individuales de un arreglo especificando el valor de un índice.

## PRÁCTICA DE AUTOEVALUACIÓN

**13.2** Dada la declaración:

```
int[] tabla = new int[3];
```

¿qué tan largo es el arreglo y cuál es el rango válido de índices?

Con frecuencia necesitamos hacer referencia al  $n$ -ésimo elemento de un arreglo, en donde  $n$  es una variable. Ésta es la forma en que podemos ver el verdadero poder de los arreglos. Por ejemplo,

suponga que deseamos sumar todos los números de un arreglo. Vamos a suponer que tenemos un arreglo con siete elementos que contienen el número de computadoras vendidas en una tienda durante cada día de la semana:

```
int[] venta = new int[7];
```

Vamos a insertar valores en el arreglo mediante instrucciones de asignación. Suponga que el lunes (día 0) se vendieron 13 computadoras:

```
venta[0] = 13;
```

y para los demás días, se vendieron las siguientes computadoras:



Ahora queremos obtener las ventas totales de la semana. La manera burda de totalizar las ventas sería escribir lo siguiente:

```
suma = venta[0] + venta[1] + venta[2] + venta[3]
 + venta[4] + venta[5] + venta[6];
```

lo cual es correcto, pero no aprovecha la regularidad de un arreglo. La alternativa es usar un ciclo **for**. Se utiliza una variable, por ejemplo **númeroDía**, para guardar el valor del índice que representa el día de la semana. Al principio el índice se establece en 0 y después se incrementa cada vez que se repite el ciclo:

```
int suma = 0;
for (int númeroDía = 0; númeroDía <= 6; númeroDía++) {
 suma = suma + venta[númeroDía];
}
```

Cada vez que se repite el ciclo, se suma el siguiente valor del arreglo al total. En realidad este fragmento de programa no es más corto que la alternativa a la que sustituye, ¡pero sería considerablemente más corto si el arreglo tuviera 1000 elementos que sumar! La otra ventaja es que el ciclo **for** muestra de manera explícita que está realizando una operación sistemática en un arreglo.

Los índices son una parte de la programación en la que se permite (algunas veces) el uso de un nombre que sea un poco enigmático. Sin embargo, en el fragmento de programa anterior queda claro el uso de **númeroDía** como índice y se relaciona en forma estrecha con el problema que se va a resolver.

## PRÁCTICA DE AUTOEVALUACIÓN

13.3 ¿Qué hace el siguiente fragmento de programa?

```
int[] tabla = new int[10];
for (int índice = 0; índice <= 10; índice++) {
 tabla[índice] = índice;
}
```

### ● La longitud de un arreglo

Un programa en ejecución siempre conoce la longitud de un arreglo. Por ejemplo, si tenemos un arreglo declarado de la siguiente forma:

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

podemos acceder a su longitud si utilizamos la propiedad `length`, como en el siguiente ejemplo:

```
int tamaño;
tamaño = tabla.length;
```

En este caso, `tamaño` tiene el valor de 10.

La propiedad `length` es una característica especial de Java. Es muy parecida a una variable `public` dentro de la clase `Array`, la cual nos permite acceder al valor del tamaño de un arreglo. Podemos obtener el valor pero no podemos modificar el valor de `length`. Cabe mencionar que no se colocan paréntesis () después de la palabra `length`, como se haría si fuera un método.

Tal vez le parezca extraño querer saber la longitud de un arreglo; después de todo, tenemos que proporcionar la longitud a la hora de declarar el arreglo. Pero más adelante veremos que esta herramienta es muy útil.

Una vez que creamos un arreglo, su longitud es fija. Los arreglos no están hechos de elástico (un arreglo no se expandirá según sea necesario para guardar información), por lo cual es imprescindible crear un arreglo que sea lo bastante grande para satisfacer las necesidades de un programa específico. Pero podemos reutilizar una variable de arreglo si creamos un nuevo arreglo de un tamaño distinto mediante `new`. Sin embargo, perderemos los datos del arreglo original.

Al diseñar un nuevo programa debemos tener en cuenta el tamaño de cualquier arreglo. Algunas veces esto es obvio con base en la naturaleza del problema. Por ejemplo, si los datos se relacionan con los días de la semana, entonces sabemos que el arreglo necesitará siete elementos. Sin embargo, hay otros tipos de estructuras de datos (como un objeto `ArrayList`) que se expanden y contraen pieza por pieza, según sea necesario.

### ● Cómo pasar arreglos como parámetros

Como vimos en capítulos anteriores del libro, los métodos son muy importantes en la programación. Un aspecto fundamental del uso de los métodos es la acción de pasar información a un método en forma de parámetros y de regresar un valor. Ahora explicaremos cómo pasar arreglos.

Suponga que queremos escribir un método cuyo trabajo sea calcular la suma de los elementos en un arreglo de enteros. Siendo programadores perceptivos, queremos que el método sea de propósito general para que pueda lidiar con arreglos de cualquier longitud. Pero eso está bien, ya que el método puede determinar fácilmente la longitud del arreglo. Entonces, el parámetro que se debe pasar al método es simplemente el arreglo y el resultado que se va a devolver al usuario del método es un número, la suma de los valores.

El siguiente es un ejemplo de cómo llamar a este método:

```
int[] tabla = new int[24];
int total;

total = sumar(tabla);
```

El método en sí sería:

```
private int sumar(int[] arreglo) {
 int total = 0;
 for (int índice = 0; índice < arreglo.length; índice++) {
 total = total + arreglo[indice];
 }
 return total;
}
```

Observe que en el encabezado del método el parámetro se declara como un arreglo, con corchetes. Pero no hay ningún parámetro que indique la longitud que debe tener éste. Para averiguar su tamaño, el método utiliza la propiedad `length`, como vimos antes. Como puede aceptar un arreglo de cualquier longitud, este método es de propósito general y puede ser muy útil. Esto es mucho más conveniente que un método de propósito específico que sólo funcione cuando el arreglo sea, por decir, de ocho elementos.

## PRÁCTICA DE AUTOEVALUACIÓN

- 13.4** Escriba un método que muestre un arreglo de enteros, uno por línea, en un área de texto. El único parámetro para el método debe ser el arreglo.

### ● La instrucción `for` mejorada

Es muy común utilizar instrucciones `for` en conjunto con los arreglos. Pero en esas ocasiones en que el programa necesita procesar cada uno de los elementos en un arreglo, hay una excelente forma de hacerlo mediante la instrucción `for` mejorada. Podemos replantear el método anterior para calcular la suma de los enteros de un arreglo como se muestra a continuación:

```
private int sumar(int[] arreglo) {
 int total = 0;
 for (int entero : arreglo) {
 total = total + entero;
 }
 return total;
}
```

Este método es más eficiente y corto. Podemos leer la instrucción `for` como “para todos los enteros `entero` en el arreglo”. El carácter de dos puntos se puede leer como “en”. El ciclo se repite para todos los elementos del arreglo. En cada repetición, la variable `entero` contiene el valor del elemento en el arreglo.

El tipo que se declara como parte de la instrucción `for` (`int` en este ejemplo) debe coincidir con el tipo que contiene el arreglo. La variable (`entero` en este ejemplo) puede tener cualquier nombre, al igual que cualquier otra variable.

Es importante recordar tres cosas a la hora de contemplar el uso de la instrucción `for` mejorada:

- Sólo podemos usarla cuando necesitamos procesar todos los elementos de un arreglo.
- El valor del índice no está disponible dentro del ciclo.
- No podemos cambiar el valor de un elemento del arreglo mediante la instrucción `for` mejorada.

Por estas razones, sólo una pequeña cantidad de programas en este capítulo utilizan la instrucción `for` mejorada.

## Uso de constantes con los arreglos

En un programa con varios arreglos se incluyen las declaraciones de éstos y también es muy probable que haya varios ciclos `for`. Los arreglos, junto con sus longitudes, se pasan al programa como parámetros. Hay muchas probabilidades de confusión, en especial si dos arreglos distintos tienen la misma longitud.

Suponga, por ejemplo, que vamos a escribir un programa para analizar las calificaciones que obtienen 10 estudiantes en sus tareas. Queremos un arreglo para guardar la calificación promedio de cada estudiante:

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

Por casualidad, hay también 10 cursos y queremos un segundo arreglo para guardar la calificación promedio en cada curso:

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

El problema es que cada vez que vemos el número 10 en el programa, no sabemos si es el número de estudiantes o el número de cursos. En este caso en particular no importa, ¡ya que son iguales!

Pero suponga que necesitamos modificar el programa para que trabaje con 20 estudiantes. Sería muy conveniente poder cambiar cada ocurrencia del número 10 por el número 20 utilizando la función “reemplazar” dentro de un editor de texto. Pero como los arreglos tienen la misma longitud, el programa podría resultar muy dañado.

Una manera de hacer más claro este programa sería declarar las longitudes de los arreglos como constantes, y después utilizar estas constantes en ciclos `for`, por ejemplo:

```
final int estudiantes = 20;
final int cursos = 24;
```

Y luego podemos utilizar las constantes de la siguiente manera:

```
int[] califEstudiante = new int[estudiantes];
int[] califCurso = new int[cursos];

for (int índice = 0; índice < estudiantes; índice++) {
 // cuerpo del ciclo
}
```

Ahora podemos realizar cambios en el programa sin problemas, al cambiar un solo número en la declaración de las constantes.

## PRÁCTICA DE AUTOEVALUACIÓN

**13.5** Escriba el código para colocar ceros en cada elemento del arreglo `califCurso`.

### Cómo inicializar un arreglo

Inicializar significa dar un valor inicial a una variable. Si usted escribe lo siguiente:

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

entonces se establece un arreglo en memoria que contiene ceros. Cuando el programador no proporciona valores iniciales de manera explícita, el compilador inserta valores predeterminados. Éstos son ceros para los números, "" para las cadenas de texto y `null` para los objetos.

Una forma común de inicializar un arreglo de manera explícita es hacerlo al momento de declararlo. Los valores iniciales requeridos se encierran entre llaves y se separan mediante comas. Pero el tamaño del arreglo **no** se debe proporcionar en su posición usual. La siguiente inicialización:

```
int[] edades = {23, 54, 96, 13, 7, 32};
```

en donde la longitud del arreglo no se da de manera explícita, es equivalente a:

```
int[] edades = new int[6];
edades[0] = 23;
edades[1] = 54;
edades[2] = 96;
edades[3] = 13;
edades[4] = 7;
edades[5] = 32;
```

He aquí otro ejemplo en el que se inicializa un arreglo de cadenas de texto:

```
String[] grupo = {"John", "Paul", "George", "Ringo"};
```

Otra forma de inicializar un arreglo es mediante un ciclo, como en el siguiente ejemplo:

```
int[] tabla = new int[25];
for (int índice = 0; índice < tabla.length; índice++) {
 tabla[índice] = 0;
}
```

Si el programa necesita restablecer periódicamente el arreglo de vuelta a sus valores iniciales, entonces podemos utilizar el ciclo `for` anterior.

## PRÁCTICA DE AUTOEVALUACIÓN

- 13.6** Declare un arreglo de cinco enteros llamado `números` y llénelo con los números del 1 al 5 como parte de la declaración.

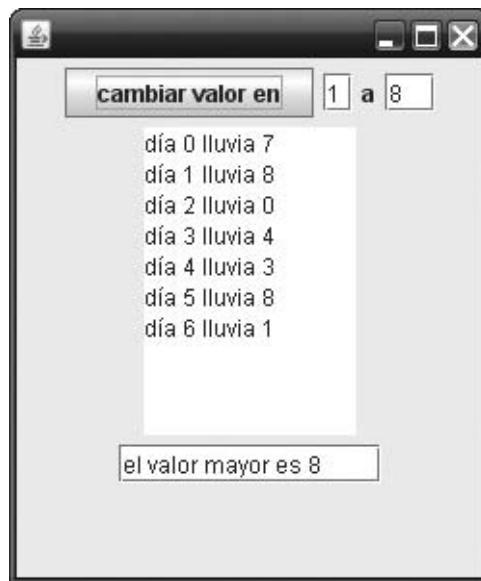
## Un programa de ejemplo

Ahora vamos a combinar todas las cosas que hemos explicado para formar un programa que recibe varios números, los coloca en un arreglo y luego los muestra en pantalla. En la figura 13.5 aparece la pantalla de este programa. Los datos desplegados representan la precipitación pluvial de los siete días en una semana (de lunes a domingo). El usuario del programa introduce el número de un día en un campo de texto y un valor de precipitación pluvial en otro campo de texto. Se muestra el mayor de los valores de precipitación pluvial.

Primero se declara el arreglo en la parte superior de la clase, para que lo puedan usar todos los métodos. Sus valores se inicializan con una selección de valores:

```
private int[] lluvia = { 7, 8, 0, 4, 3, 8, 1 };
```

Luego declaramos el código para mostrar los valores del arreglo en un área de texto:



**Figura 13.5** El programa de precipitación pluvial.

```
private void mostrar() {
 datos.setText("");
 for (int númeroDía = 0; númeroDía <= 6; númeroDía++) {
 datos.append("día " + Integer.toString(númeroDía)
 + " lluvia " + Integer.toString(lluvia[númeroDía])
 + "\n");
 }
}
```

A continuación veremos el código para colocar un nuevo valor en un elemento del arreglo. El valor del índice está en un cuadro de texto; el valor de los datos está en otro. Al último se invoca el método **mostrar** para desplegar el valor actualizado y después se invoca el método **mayor** para mostrar el valor más grande:

```
private void nuevoValor() {
 int índice;
 int datos;
 índice = Integer.parseInt(día.getText());
 datos = Integer.parseInt(cantidad.getText());
 lluvia[índice] = datos;
 mostrar();
 mayor();
}
```

Ahora veremos el código para calcular el mayor valor de precipitación pluvial. La metodología que utilizamos es empezar suponiendo que el primer elemento es el mayor. Despues analizamos el resto de los elementos en turno, comparándolos con este valor mayor. Si encontramos un valor que sea mayor que el que ya tenemos, actualizamos nuestro valor mayor. Ésta es una metodología clásica.

```
private void mayor() {
 int másalto;

 másalto = lluvia[0];
 for (int índice = 0; índice <= 6; índice++) {
 if (másalto < lluvia[índice]) {
 másalto = lluvia[índice];
 }
 }
 cifras.setText("el valor mayor es " + Integer.toString(másalto));
}
```

También podríamos replantear este método de la siguiente manera, mediante la instrucción **for** mejorada:

```
private void mayor() {
 int másalto;

 másalto = lluvia[0];
 for (int precipitación : lluvia) {
 if (másalto < precipitación) {
 másalto = precipitación;
 }
 }
}
```

Cabe mencionar que es muy común utilizar la instrucción **for** junto con los arreglos. Desde luego que esto se debe a que un ciclo **for** aprovecha al máximo la uniformidad de los arreglos.

## PRÁCTICA DE AUTOEVALUACIÓN

**13.7** Escriba un método para calcular y mostrar la precipitación pluvial total de la semana.

### Búsqueda rápida (lookup)

Parte del poder de los arreglos es que podemos buscar algo con mucha facilidad y rapidez. En el programa de la precipitación pluvial podemos extraer el valor de la precipitación pluvial del Martes con sólo hacer referencia a `lluvia[1]`. Lo mismo se aplica a cualquier información a la que se pueda hacer referencia mediante un índice entero. Por ejemplo, si tenemos una tabla que muestre

la altura promedio de las personas de acuerdo con su edad, podemos indizar la tabla usando una edad (25 en este ejemplo):

```
double[] altura = new double[100];
double miAltura;
miAltura = altura[25];
```

De manera similar, si hemos enumerado los días de la semana del 1 al 6, podemos convertir un número en una cadena de texto mediante el siguiente código:

```
int númeroDía;
string nombreDía;
string[] nombre = {"Lunes", "Martes", "Miércoles", "Jueves",
 "Viernes", "Sábado", "Domingo"};
nombreDía = nombre[númeroDía];
```

También podríamos lograr esto mediante una instrucción `switch`, pero el código sería más extenso y probablemente más complicado.

El proceso de usar un arreglo para buscar algo es extremadamente útil, simple y aprovecha el poder de los arreglos.

## PRÁCTICA DE AUTOEVALUACIÓN

**13.8** Vuelva a escribir la conversión anterior utilizando una instrucción `switch`.

### Búsqueda detallada (search)

Otra forma de acceder a la información en un arreglo es buscándola. Esto es lo que los humanos hacemos con un directorio telefónico o un diccionario. El ejemplo que consideraremos a continuación es un directorio telefónico (figura 13.6).

Vamos a establecer dos arreglos, uno para guardar nombres y el otro para guardar los números telefónicos correspondientes:

```
private String[] nombres = new String[20];
private String[] números = new String[20];
```



**Figura 13.6** El directorio telefónico.

Ahora que hemos creado los arreglos, podemos colocar datos en ellos:

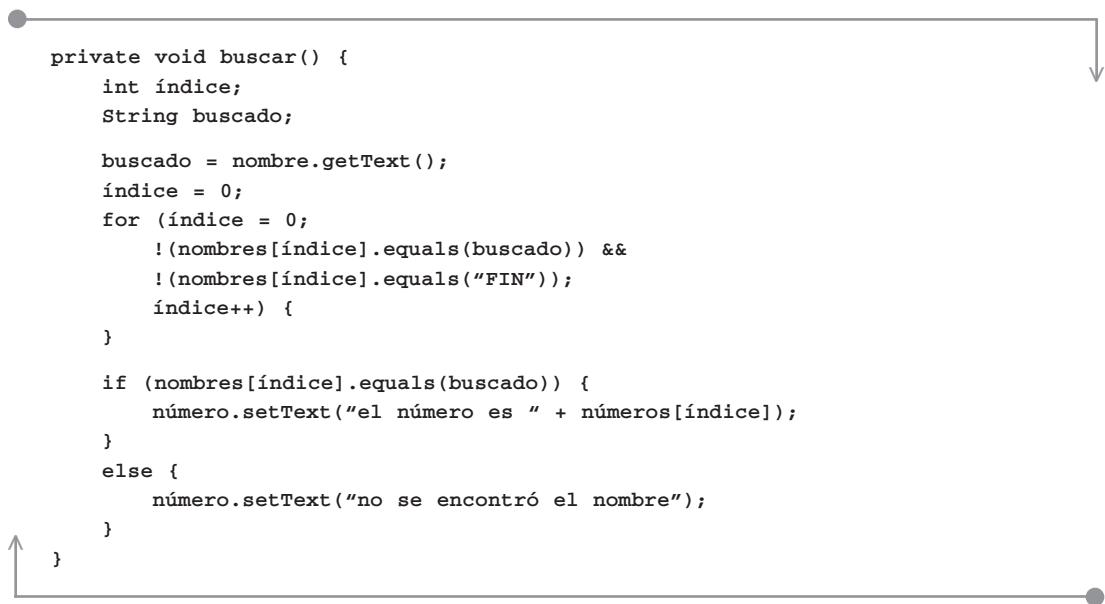
```
 nombres[0] = "Alex";
 números[0] = "2720774";

 nombres[1] = "Megan";
 números[1] = "5678554";

 nombres[2] = "FIN";
```

Una forma simple y efectiva de buscar en el directorio es empezar desde el principio y recorrer una entrada tras otra hasta llegar al nombre que estamos buscando. Sin embargo, el nombre que buscamos podría no estar en el directorio, y debemos tener en cuenta la posibilidad de que surja dicha situación. Entonces la búsqueda continúa hasta encontrar lo que buscamos o hasta llegar al final de las entradas. Podríamos comprobar que hayamos llegado al final del arreglo, pero una metodología más conveniente es colocar una entrada especial en el arreglo para indicar el final de los datos útiles. Este marcador final consistirá en una entrada con el nombre **FIN**.

Ahora podemos escribir el ciclo para buscar el número telefónico que deseamos:



```
private void buscar() {
 int índice;
 String buscado;

 buscado = nombre.getText();
 índice = 0;
 for (índice = 0;
 !(nombres[índice].equals(buscado)) &&
 !(nombres[índice].equals("FIN"));
 índice++) {
 }

 if (nombres[índice].equals(buscado)) {
 número.setText("el número es " + números[índice]);
 }
 else {
 número.setText("no se encontró el nombre");
 }
}
```

A esto se le conoce como búsqueda *serial o secuencial*. Empieza al principio del arreglo con el índice cero y continúa buscando elemento por elemento, agregando 1 al índice. La búsqueda continúa hasta encontrar el elemento deseado o hasta llegar al nombre especial **FIN**.

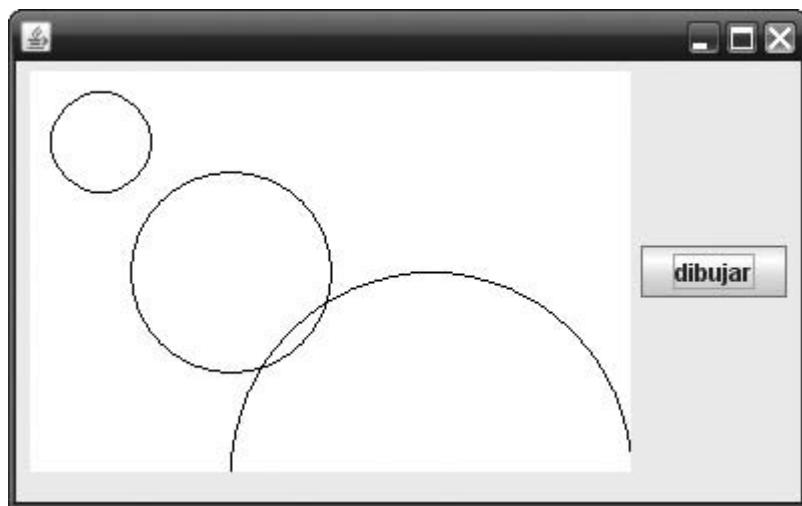
Este tipo de búsqueda no hace ninguna suposición en cuanto al orden de los elementos en la tabla; pueden estar en cualquier orden. Otras técnicas de búsqueda explotan el orden de los elementos en una tabla, como el orden alfabético. Estas técnicas están más allá del alcance de este libro.

Por lo general, la información, como los números telefónicos, se almacena en un archivo en vez de hacerlo en un arreglo, ya que los datos que se guardan en un archivo son más permanentes. Comúnmente se busca en el archivo la información requerida en vez de hacerlo en un arreglo.

También se puede introducir el archivo en la memoria, para después guardar la información en un arreglo y realizar la búsqueda anterior.

## Arreglos de objetos

Los arreglos pueden guardar cualquier cosa: enteros, números de punto flotante, cadenas de texto, botones, controles deslizables, cualquier objeto en la biblioteca o cualquier objeto que construya el programador. La única restricción es que todos los objetos en un arreglo deben ser del mismo tipo. En el siguiente ejemplo vamos a crear un arreglo de objetos globo (figura 13.7). Ya habíamos visto el objeto globo en capítulos anteriores del libro.



**Figura 13.7** Cómo mostrar un arreglo de globos.

Un objeto globo (que en realidad es sólo un círculo) tiene un tamaño y una posición en la pantalla. Se proporcionan métodos como parte del objeto para moverlo, cambiar su tamaño y mostrarlo en pantalla. He aquí la descripción de la clase:

```
import java.awt.*;
public class Globo {
 private int x;
 private int y;
 private int diámetro;
 public Globo(int xInicial, int yInicial,
 int diámetroInicial) {
 x = xInicial;
 y = yInicial;
 diámetro = diámetroInicial;
 }
}
```

```

 public void cambiarTamaño(int cambio) {
 diámetro = diámetro + cambio;
 }

 public void mostrar(Graphics papel) {
 papel.drawOval(x, y, diámetro, diámetro);
 }
}

```

Ahora podemos crear un arreglo de globos:

```
private Globo[] fiesta = new Globo[10];
```

Pero esta instrucción sólo crea el arreglo, listo para guardar todos los globos. Ahora necesitamos crear algunos globos, como se muestra a continuación:

```

fiesta[0] = new Globo(10, 10, 50);
fiesta[1] = new Globo(50, 50, 100);
fiesta[2] = new Globo(100, 100, 200);

```

y mostrar todos los globos:

```

private void mostrarGlobos(Graphics papel) {
 for (int g = 0; g <= 2; g++) {
 fiesta[g].mostrar(papel);
 }
}

```

La ventaja de almacenar los objetos globo en un arreglo es que podemos hacer algo con todos ellos de una manera conveniente. Por ejemplo, podemos cambiar el tamaño de todos los globos a la vez:

```

private void cambiarTamaño() {
 for (int g = 0; g <= 2; g++) {
 fiesta[g].cambiarTamaño(20);
 }
}

```

Para terminar, ya habíamos dicho que todos los elementos del arreglo tienen que ser del mismo tipo. Hay una excepción: si declara un arreglo de objetos de la clase `Object`, entonces puede colocar distintos tipos de objetos en el arreglo. Esto se debe a que `Object` es la superclase de todas las demás clases.

## Fundamentos de programación

Un arreglo es una colección de elementos de datos con un solo nombre. Todos los elementos de un arreglo son del mismo tipo. Los elementos individuales de un arreglo se identifican mediante un índice, un entero. Por ejemplo, si tenemos un arreglo llamado `tabla`, podemos hacer referencia a uno de sus elementos individuales como `tabla[2]`, en donde 2 es el índice. Podemos hacer referencia de manera similar a un elemento de un arreglo utilizando una varia-



*Fundamentos de programación (continúa)*

ble entera como el índice; por ejemplo, `tabla[índice]`. Esta herramienta es la que hace a los arreglos poderosos.

Una vez creado, un arreglo tiene una longitud que permanece fija.

Los arreglos pueden guardar datos de cualquier tipo; por ejemplo, `int`, `double`, `boolean`, `JButton`, `JTextField` (pero en cualquier arreglo, todos los datos deben ser del mismo tipo).

El arreglo es la estructura de datos más antigua y utilizada. Son compactos y se puede acceder a ellos con mucha rapidez mediante instrucciones del hardware de la computadora.

Es común utilizar el ciclo `for` junto con los arreglos.

Al comprender los arreglos podemos tener una imagen completa de los programas que presentamos en este libro. La mayoría de los programas incluyen la siguiente instrucción:

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

Éste es el encabezado del método que el sistema llama al momento de iniciar una aplicación. Debe tener el nombre `main`. El método tiene un solo parámetro llamado `args` por convención. Como podemos ver, este parámetro es un arreglo de cadenas de texto. Su tamaño depende de la función que desempeñe el programa. A menudo, como es el caso con la mayoría de los programas de este libro, el arreglo es de longitud cero. Al llamar a `main` se le provee un arreglo de cadenas de texto. Esto permite iniciar un programa (por lo general desde la línea de comandos) con información apropiada para guiar su comportamiento. Consulte el capítulo 17 para obtener más detalles.

En el capítulo 12 analizamos los objetos `ArrayList`. Estos objetos son una estructura de datos que se constituye como rival para los arreglos. En la siguiente tabla sintetizamos las diferencias:

|                                                                                                                | <b>Arreglo</b> | <b>Objeto <code>ArrayList</code></b> |
|----------------------------------------------------------------------------------------------------------------|----------------|--------------------------------------|
| Su tamaño se declara al momento de su creación                                                                 | sí             | no                                   |
| Se expande y contrae según sea necesario                                                                       | no             | sí                                   |
| Se pueden insertar y eliminar elementos desde el interior de la estructura                                     | no             | sí                                   |
| Puede alojar cualquier objeto                                                                                  | sí             | sí                                   |
| Puede alojar valores <code>int</code> , <code>double</code> , <code>boolean</code> sin necesidad de conversión | sí             | no                                   |
| Velocidad                                                                                                      | más rápido     | más lento                            |

## Errores comunes de programación

Un error común en Java es confundir la longitud de un arreglo con el rango de índices válidos. Por ejemplo, el arreglo:

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

tiene 10 elementos. El rango válido de índices para este arreglo es de 0 a 9. La referencia a `tabla[10]` es una referencia a un elemento del arreglo que simplemente no existe. Por fortuna el sistema de Java comprueba este tipo de violaciones al momento de ejecutar el programa, por lo que emitirá un mensaje de error.

He aquí un ejemplo común de cómo hacer las cosas mal:

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

for (int índice = 0; índice <= 10; índice++) { // advertencia, hay un error
 tabla[índice] = 0;
}
```

Esto colocará un cero en todos los elementos del arreglo **tabla**, pero después se colocará un cero en el elemento de datos que se encuentre inmediatamente después del arreglo en la memoria de la computadora. En este caso el programa falla con un mensaje **ArrayIndexOutOfBoundsException** (excepción de índice de arreglo fuera de rango). Siempre es conveniente comprobar cuidadosamente la condición para terminar un ciclo **for** cuando lo utilizamos con un arreglo.

Algunas veces los estudiantes tienen dificultades para visualizar en dónde está un arreglo. Éste se guarda en la memoria principal; es invisible y sólo tiene vida cuando el programa se ejecuta.

La instrucción **for** mejorada puede hacer que un ciclo sea más conciso. Pero tenga en cuenta lo siguiente:

- Sólo puede usar la instrucción **for** mejorada cuando necesita procesar todos los elementos de un objeto **ArrayList**;
- El valor del índice no está disponible dentro del ciclo;
- No puede modificar el valor de un elemento del arreglo mediante la instrucción **for** mejorada.

## Secretos de codificación

Un arreglo con 20 elementos se declara de la siguiente forma:

```
double[] tabla = new double[20];
```

Para hacer referencia a un elemento del arreglo, el índice se escribe entre corchetes, como en el siguiente ejemplo:

```
tabla[3] = 12.34;
```

## Resumen

Un arreglo es una colección de datos. El programador le asigna un nombre. Todos los elementos de un arreglo deben ser del mismo tipo (por ejemplo, todos **int**).

Un arreglo se declara (al igual que otras variables) de la siguiente forma:

```
int[] harry = new int[25];
```

El arreglo tiene 25 elementos. El valor del índice más grande es 24.

Para hacer referencia a un elemento individual de un arreglo utilizamos un índice entero, por ejemplo:

```
harry[12] = 45;
```

Los índices tienen valores que empiezan en cero y llegan hasta el valor del índice más grande.

## Ejercicios

### Juegos

- 13.1 Nim** El humano juega contra la computadora. Al inicio del juego hay tres pilas de cerillos. En cada pila hay un número aleatorio de cerillos en el rango de 1 a 20. Las tres pilas se muestran en pantalla durante todo el juego. Una opción aleatoria determina quién va primero. Los jugadores toman turnos para extraer todos los cerillos que deseen de cualquier pila, pero sólo de una. Un jugador debe extraer por lo menos un cerillo. El ganador es quien haga que el otro jugador tome el último cerillo. Haga que la computadora juegue al azar; es decir, que seleccione una pila al azar y después una cantidad aleatoria de cerillos con base en los que haya disponibles.
- 13.2 Combinación de bóveda** Establezca un arreglo que contenga los seis dígitos para abrir una bóveda de seguridad. Pida al usuario que introduzca seis dígitos, uno a la vez, mediante botones etiquetados con los dígitos del 0 al 9, y compruebe que sean correctos. Al introducir un dígito indique al usuario si es correcto o no; el usuario tendrá tres oportunidades antes de hacerlo empezar desde cero otra vez.
- 13.3 Blackjack** (conocido también como Veintiuno, *vingt-et-un* o Pontoon) Escriba un programa para jugar a este juego de cartas. La computadora debe actuar como el repartidor. Al principio el repartidor le debe entregar dos cartas, las cuales deben ser aleatorias (en el juego real el repartidor tiene una enorme cantidad de cartas provenientes de varios mazos barajados). Su objetivo es obtener una puntuación más alta que la del repartidor, sin pasar de 21. El as cuenta como 1 o como 11. En cualquier momento usted puede “pedir”, lo cual significa que desea otra carta, o “plantarse”, lo cual significa que está conforme con lo que tiene. También puede “pasarse”, lo cual significa que tiene más de 21. Cuando se planta o se pasa, es turno del repartidor para repartirse cartas a sí mismo. El objetivo del repartidor es obtener una puntuación mayor a la de usted, sin pasarse. Pero el repartidor no conoce su puntuación y por ello se arriesga a tratar de mejorar su puntuación.

Proporcione botones para empezar un nuevo juego, pedir y plantarse. Al final de un juego muestre los dos conjuntos de cartas que se repartieron y los totales.

### Operaciones básicas con arreglos

- 13.4 Datos pluviales** Complete el programa para manejar los datos de precipitación pluvial; incluya las siguientes operaciones:
- Sumar los valores y mostrar el total.
  - Buscar el valor más pequeño y mostrarlo en pantalla.
  - Buscar el índice del valor más grande.
- 13.5 Arreglo de cadenas** Escriba un programa que utilice un arreglo de 10 cadenas de texto. Cada elemento del arreglo debe contener una palabra. Escriba métodos que realicen cada una de las siguientes operaciones:
- Introducir valores con el teclado mediante un campo de texto.
  - Mostrar todas las palabras (ahora puede observar que se hayan introducido correctamente en el arreglo).
  - Introducir una palabra mediante un campo de texto y buscarla para ver si está presente en el arreglo. Mostrar un mensaje que diga si la palabra está presente o no.

- 13.6 Gráfico de barras** Los gráficos de barras son útiles para los datos como la precipitación pluvial. Escriba un programa que muestre un gráfico de barras de los datos que reciba mediante un arreglo. El arreglo debe guardar varios valores sobre la precipitación pluvial en cada uno de los siete días de la semana. Puede utilizar el método `fillRect` de la biblioteca para dibujar las barras individuales.
- 13.7 Gráfico de pastel** Los gráficos de pastel muestran las proporciones de las cantidades y son por lo tanto útiles para datos como presupuestos personales o empresariales. Escriba un programa que muestre en pantalla un gráfico de pastel de los datos que reciba mediante un arreglo. El arreglo debe guardar las cantidades invertidas en, por ejemplo, viajes, alimentos, alojamiento, etc. Investigue el método `fillArc` en la clase `Graphics` de la biblioteca de Java, el cual le será muy útil en este programa. Cree un arreglo que guarde los colores disponibles (`Color.black`, `Color.lightGray`, `Color.blue`, etc.) y utilícelo para colorear las distintas rebanadas del pastel.
- 13.8 Trazador de gráficos** Escriba un método para dibujar un gráfico de datos que se proporcionan mediante un arreglo de coordenadas *x* y un arreglo de las correspondientes coordenadas *y*. El método debe tener el siguiente encabezado:

```
private void dibujarGráfico(double[] x, double[] y)
```

Debe dibujar líneas rectas de un punto a otro. También debe dibujar los ejes.

## Estadísticas

- 13.9 Suma y media** Escriba un programa que introduzca una serie de enteros en un arreglo. Los números deben estar en el rango de 0 a 100.  
Calcule y muestre en pantalla:

- El número más grande.
- El número más pequeño.
- La suma de los números.
- La media de los números.

Muestre un histograma (gráfico de barras) que indique cuántos números se encuentran en los rangos de 0 a 9, de 10 a 19, etc.

## Números aleatorios

- 13.10 Verifique la clase generadora de números aleatorios** Verifique que la clase generadora de números aleatorios (capítulo 6) funcione correctamente. Configúrela para que proporcione números aleatorios en el rango de 1 a 100. Después invoque el método 100 veces y coloque las frecuencias en un arreglo, como en el ejercicio anterior. Por último, muestre en pantalla el histograma de frecuencias, de nuevo como en el ejercicio anterior. Los números aleatorios deben ser aleatorios, por lo que el histograma deberá tener barras con una altura aproximadamente igual.

## Palabras

- 13.11 Permutación de palabras** Escriba un programa que reciba como entrada cuatro palabras y después muestre en pantalla todas las posibles permutaciones de éstas. Por ejemplo, si se introducen las palabras “malo”, “perro”, “muerde” y “hombre”, entonces debe aparecer lo siguiente en pantalla:

hombre muerde perro malo  
 hombre malo muerde perro  
 perro muerde hombre malo  
 etc.

(¡No todas las oraciones deben tener sentido!).

## Procesamiento de información —búsqueda

- 13.12 Diccionario** Establezca dos arreglos que contengan pares de palabras equivalentes en inglés y español. Después introduzca una palabra en inglés, busque su equivalente en español y muestre el resultado en pantalla. Asegúrese de comprobar que la palabra esté en el diccionario. Después agregue la herramienta para traducir en sentido opuesto, utilizando los mismos datos.
- 13.13 Biblioteca** Cada uno de los miembros de una biblioteca tiene un código de usuario único, un entero. Cuando alguien desea pedir un libro prestado, se comprueba que su código de usuario sea válido. Escriba un programa que busque un código de usuario específico dentro de una tabla de códigos de usuario. El programa debe mostrar un mensaje que indique si el código es o no válido.
- 13.14 Directorio telefónico** Mejore el programa del directorio telefónico que vimos antes en este capítulo, de manera que se puedan agregar nuevos nombres y números al directorio. Después agregue una herramienta para eliminar un nombre y un número.

## Procesamiento de información —ordenamiento

- 13.15 Ordenamiento** Escriba un programa que reciba como entrada una serie de números, los ordene en forma numérica ascendente y los muestre en pantalla.  
 Este programa no es fácil de escribir. Hay varias metodologías para ordenar datos; de hecho hay libros enteros sobre el tema. Una de las metodologías más comunes es la siguiente.  
 Busque el número más pequeño en el arreglo. Intercámbielo con el primer elemento del arreglo. Ahora el primer elemento está en la posición correcta. Deje este primer elemento donde está y repita la operación con el resto del arreglo (todos los elementos excepto el primero). Repita la operación con un arreglo cada vez más pequeño hasta que esté completamente ordenado.

## Arreglos de objetos

- 13.16 Globos** Extienda el programa que mantiene un arreglo de globos. Agregue herramientas para:
- Reventar todos los globos con base en un factor aleatorio.
  - Mover todos los globos con base en el mismo factor.
- 13.17 Directorio telefónico** Escriba un programa para crear y mantener un directorio telefónico. Cada elemento del arreglo debe ser un objeto de la clase **Entrada**:

```
public class Entrada {
 private String elNombre;
 private String elNúmero;
 // métodos para acceder al nombre y al número
}
```

Complete la clase **Entrada** y después cree el siguiente arreglo:

```
private Entrada[] directorio = new Entrada[1000];
```

ahora coloque datos en ella como se muestra a continuación, utilizando los métodos:

```
directorio[0].setNombre("Douglas Bell");
directorio[0].setNúmero("01 0114 255 3103");
```

Proporcione una GUI para introducir datos en el directorio. Provea una herramienta de búsqueda, de manera que si se introduce un nombre en un cuadro de texto aparezca el correspondiente número telefónico.

- 13.18 Juego de cartas** Éste es un ejemplo que podría ser parte de un juego de cartas. Cada carta se describe mediante la siguiente clase:

```
public class Carta {
 private int rango;
 private String palo;
 // métodos para acceder al rango y al palo
}
```

Complete la clase **Carta**. Después cree un arreglo para guardar un mazo completo de cartas:

```
private Carta[] mazo = new Carta[52];
```

Inicialice el mazo mediante un ciclo **for** para recorrer los cuatro palos y un ciclo **for** anidado para recorrer los distintos rangos de las cartas.

## Respuestas a las prácticas de autoevaluación

**13.1**    `int[] precipitación = new int[7];`

**13.2** El arreglo tiene tres elementos de longitud. Los índices válidos son de 0 a 2.

**13.3** El fragmento de programa coloca los números del 0 al 10 en el arreglo. Pero trata de acceder a un elemento inexistente con el valor de índice 10. Por lo tanto, el programa fallará.

**13.4**    `private void mostrar(int[] arreglo) {
 áreaTexto.setText("");
 for (int índice = 0; índice < arreglo.length; índice++) {
 áreaTexto.append(Integer.toString(arreglo[índice]) + "\n");
 }
}`

**13.5**    `for (int índice = 0; índice < cursos; índice++) {
 califCurso[índice] = 0;
}`

**13.6**    `int[] números = {1, 2, 3, 4, 5};`





Respuestas a las prácticas de autoevaluación (continúa)

13.7     private void totalSemana() {  
             int total = 0;  
  
             for (int índice = 0; índice <= 6; índice++ ) {  
                 total = total + lluvia[índice];  
             }  
             cifras.setText("el total es " + Integer.toString(total));  
    }

O, si utilizamos la instrucción **for** mejorada:

```
private void totalSemana() {
 int total = 0;

 for (int precipitación : lluvia) {
 total = total + precipitación;
 }
 cifras.setText("el total es " + Integer.toString(total));
}

13.8 switch (númeroDía) {
 case 0:
 nombreDía = "Lunes";
 break;
 case 1:
 nombreDía = "Martes";
 break;
 case 2:
 nombreDía = "Miércoles";
 break;
 case 3:
 nombreDía = "Jueves";
 break;
 case 4:
 nombreDía = "Viernes";
 break;
 case 5:
 nombreDía = "Sábado";
 break;
 case 6:
 nombreDía = "Domingo";
 break;
}
```

# CAPÍTULO 14



## Matrices

En este capítulo conoceremos cómo:

- Declarar una matriz (o arreglo bidimensional).
- Utilizar los índices con las matrices.
- Obtener el tamaño de una matriz.
- Pasar matrices como parámetros.
- Inicializar una matriz.

### ● Introducción

Las matrices o arreglos bidimensionales (también conocidas como tablas) son muy comunes en la vida diaria:

- Un tablero de ajedrez.
- El itinerario de los trenes.
- Una hoja de cálculo.

En el capítulo anterior vimos los arreglos unidimensionales. Java cuenta con una forma natural para extender los arreglos unidimensionales a dos dimensiones. Por ejemplo, la siguiente declaración:

```
int[][] ventas = new int[4][7];
```

Declara una matriz o arreglo bidimensional de enteros. Contiene las cifras de las ventas de computadoras en cada una de cuatro tiendas para cada uno de los siete días de una semana (figura 14.1). La matriz se llama **ventas**. Podemos considerar que tiene cuatro filas y siete columnas. Cada fila representa una semana en una tienda específica. Cada columna representa un día individual en cada una de las cuatro tiendas. Los índices de las filas van de 0 a 3. Los índices de las columnas van de 0 a 6. La columna 0 es el lunes, la columna 1 es el martes, etc.

|                              |   | Números de columna (días) |    |    |    |   |    |    |
|------------------------------|---|---------------------------|----|----|----|---|----|----|
|                              |   | 0                         | 1  | 2  | 3  | 4 | 5  | 6  |
| Números de fila<br>(tiendas) | 0 | 22                        | 49 | 4  | 93 | 0 | 12 | 32 |
|                              | 1 | 3                         | 8  | 67 | 51 | 5 | 3  | 63 |
|                              | 2 | 14                        | 8  | 23 | 14 | 5 | 23 | 16 |
|                              | 3 | 54                        | 0  | 76 | 31 | 4 | 3  | 99 |

**Figura 14.1** Una matriz o arreglo bidimensional.

## PRÁCTICA DE AUTOEVALUACIÓN

- 14.1** ¿Cuál columna representa el sábado? ¿Cuántas computadoras se vendieron el jueves en la tienda 3? ¿Cuál es el número de fila y de columna para esta cifra?

### Cómo declarar una matriz

Al igual que con otras variables y objetos, para declarar una matriz utilizamos `new`, ya sea en la parte superior de la clase o de un método. El programador asigna un nombre a la matriz, como en el siguiente ejemplo:

```
int[][] ventas = new int[4][7];
double[][] temps = new double[10][24];
```

Al declarar una matriz es necesario indicar cuántas filas y columnas tiene. La matriz `ventas` tiene cuatro filas: una para cada una de cuatro tiendas. Tiene siete columnas: una para cada día de la semana. La matriz contiene cifras de ventas para cada una de las cuatro tiendas durante cada día de la semana. La matriz `temps` contiene información sobre las temperaturas en cada uno de 10 hornos, para cada hora durante un periodo de 24 horas.

Al igual que con cualquier otra variable, es común (y conveniente) elegir un nombre para la matriz que describa con claridad para qué se va a utilizar. El nombre es para la matriz completa: para toda la colección de datos.

## PRÁCTICA DE AUTOEVALUACIÓN

- 14.2** Declare una matriz para representar un tablero de ajedrez de  $8 \times 8$ . Cada posición en la matriz debe contener una cadena de texto.

## Índices

Para hacer referencia a un elemento individual en una matriz, un programa debe especificar los valores de dos índices enteros (también conocidos como subíndices). Por lo tanto, `ventas[3][2]` se refiere al elemento de la matriz que se encuentra en la fila 3 y la columna 2, lo cual representa a la tienda número 3 y el día número 2 (miércoles). De manera similar, `tableroAjedrez[2][7]` podría contener la cadena de texto “peón”.

Para introducir un valor para un elemento de una matriz podemos utilizar el siguiente código:

```
ventas[3][2] = Integer.parseInt(campoTexto.getText());
tableroAjedrez[3][4] = campoTexto.getText();
```

y de manera similar podemos mostrar los valores de los elementos de una matriz mediante cuadros de texto.

Podemos modificar los valores con instrucciones de asignación, como en el siguiente ejemplo:

```
ventas[3][2] = 99;
tableroAjedrez[2][7] = "caballo"; // coloca un caballo en uno de los cuadros
```

En todos estos fragmentos de programa, para referirnos a los elementos individuales de una matriz declaramos los valores de los índices que identifican al elemento específico en el que estamos interesados.

A menudo es necesario hacer referencia a un elemento de una matriz mediante la especificación de *variables* para cada uno de los dos índices. Ésta es la forma en la que podemos aprovechar el poder de las matrices. Como ejemplo, suponga que queremos sumar todos los números en una matriz de números que contiene datos sobre las ventas de computadoras en cuatro tiendas, durante un periodo de siete días:

```
int[][] ventas = new int[4][7];
```

La manera burda de sumar las ventas sería escribir lo siguiente:

```
suma =
 ventas[0][0] + ventas[0][1] + ventas[0][2] + ventas[0][3]
 + ventas[0][4] + ventas[0][5] + ventas[0][6]
 + ventas[1][0] + ventas[1][1] + ventas[1][2]
 + ventas[1][3] + ventas[1][4] + ventas[1][5] + ventas[1][6]
 + etcétera;
```

el código anterior es largo, difícil de entender y propenso a errores, pero es correcto. Sin embargo, no aprovecha la regularidad de una matriz. La alternativa sería utilizar un ciclo `for`. Se utilizan variables para guardar los valores de los índices. Al principio cada índice se hace igual a 0 y después se incrementa cada vez que se repite el ciclo:

```
int[][] ventas = new int[4][7];
int suma;

suma = 0;
for (int tienda = 0; tienda <= 3; tienda++) {
 for (int númeroDía = 0; númeroDía <= 6; númeroDía++) {
 suma = suma + ventas[tienda][númeroDía];
 }
}
```

lo cual es mucho más corto y ordenado que si hubiéramos escrito todas las sumas con mucho detalle.

## PRÁCTICA DE AUTOEVALUACIÓN

- 14.3 Escriba instrucciones para colocar el texto “vacío” en cada uno de los cuadros del tablero de ajedrez.

### El tamaño de una matriz

Cuando creamos una matriz de la siguiente forma:

```
double[][] info = new double[20][40];
```

tiene un tamaño fijo que no se puede modificar, a menos que volvamos a crear la matriz completa mediante `new`.

Siempre podemos obtener el tamaño de una matriz mediante la propiedad `length`. Por ejemplo, para la matriz anterior podemos usar:

```
int tamañoFila = info.length;
```

lo cual nos da un valor de 20, en tanto que:

```
int tamañoColumna = info[0].length;
```

nos da un valor de 40. Esta instrucción proporciona la longitud de la fila cero de la matriz, pero como todas las filas son del mismo tamaño, está bien hacerlo.

Cabe mencionar que **no** se colocan paréntesis ( ) después de la propiedad `length`.

## PRÁCTICA DE AUTOEVALUACIÓN

- 14.4 ¿Cuál es el valor de `tableroAjedrez.length`?

## Paso de matrices como parámetros

Suponga que necesitamos escribir un método cuya función sea calcular la suma de los elementos en una matriz de enteros. Queremos que el método sea de propósito general, para poder lidar con matrices de cualquier tamaño. Por lo tanto, debemos pasar el nombre de la matriz como parámetro al método; además, el resultado que se debe devolver al usuario del método es un número: la suma de los valores.

El siguiente es un ejemplo de una llamada al método:

```
int[][] ventas = new int[24][12];
int total;
total = sumar(ventas);
```

Y el código del método es:

```
private int sumar(int[][] matriz) {
 int total = 0;
 for (int fila = 0; fila < matriz.length; fila++) {
 for (int col = 0; col < matriz[0].length; col++) {
 total = total + matriz[fila][col];
 }
 }
 return total;
}
```

## Uso de constantes con matrices

El uso de constantes puede evitar la confusión, en especial si dos matrices distintas tienen la misma longitud. Por ejemplo, en el programa para analizar las cifras de ventas de las computadoras en varias tiendas durante varios días, utilizamos una matriz para guardar las cifras. Cada columna representa un día. Las filas son los datos para cada tienda. Ahora suponga que, por coincidencia, hay siete tiendas. La matriz sería:

```
int[][] ventas = new int[7][7];
```

El problema es que cada vez que veamos el número 7 en el programa, no sabremos si es el número de tiendas o el número de días. Desde luego que en este caso no importa, ¡ya que son iguales! Pero suponga que necesitamos modificar el programa para que trabaje con ocho tiendas. Sería muy conveniente cambiar cada ocurrencia del número 7 por el número 8 usando el editor. Pero esto es extremadamente peligroso ya que las longitudes son iguales.

Una forma excelente de clarificar el programa sería declarar los valores máximos de los índices como constantes, de la siguiente forma:

```
final int días = 7;
final int tiendas = 7;
```

y después podríamos declarar la matriz así:

```
int[][] ventas = new int[tiendas][días];
```

Ahora bien, si cambia el número de tiendas podemos realizar el ajuste correspondiente al programa con confianza, ya que sólo tenemos que modificar un número en la declaración de las constantes. También podemos escribir ciclos `for` que utilicen las constantes:

```
for (int índice = 0; índice < tiendas; índice++) {
 // cuerpo del ciclo
}
```

## Cómo inicializar una matriz

Inicializar significa asignar un valor inicial a una variable. Si escribe lo siguiente:

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

entonces se prepara espacio en la memoria para la matriz, que contiene ceros. El compilador asigna valores iniciales a las matrices que no se inicializan de manera explícita. Si la matriz consiste en números, le asigna ceros. Si consiste en cadenas de texto, le asigna el valor `""`. Si la matriz consiste en objetos, asigna el valor `null` a todos sus elementos.

Una forma de inicializar una matriz de manera explícita es utilizar ciclos anidados, como en el siguiente ejemplo:

```
for (int fila = 0; fila <= 9; fila++) {
 for (int col = 0; col <= 9; col++) {
 tabla[fila][col] = 99;
 }
}
```

Otra forma de inicializar una matriz es declararla como se muestra a continuación:

```
int[][] matriz =
 {{1, 0, 1},
 {0, 1, 0}};
```

Observe el uso de las llaves y las comas. El código anterior crea una matriz con dos filas y tres columnas, y le asigna sus valores iniciales. Cuando utilizamos esta forma de inicialización, el tamaño de la matriz **no** debe aparecer en los corchetes. La inicialización se lleva a cabo una vez, al momento de crear la matriz. Si el programa modifica el valor de un elemento en la matriz, este valor no regresará a su valor original —por lo menos no hasta que se vuelva a ejecutar el programa.

Si el programa necesita restablecer en forma periódica la matriz de vuelta a sus valores iniciales, entonces debemos utilizar ciclos `for` como vimos antes.

## PRÁCTICA DE AUTOEVALUACIÓN

- 14.5** Escriba la declaración de una matriz de cadenas de texto con una longitud de  $3 \times 3$ , de tal forma que se llene con las palabras “uno”, “dos”, “tres”, etc.

## Un programa de ejemplo

El siguiente programa mantiene una matriz de enteros. Estos valores representan la precipitación pluvial en un periodo de siete días, en cada una de tres ubicaciones. La pantalla se muestra en la figura 14.2. La matriz se despliega en un área de texto con un surtido inicial de valores. El usuario puede modificar un valor en la matriz, para lo cual debe especificar los valores de sus índices y el nuevo valor de los datos.



Figura 14.2 Matriz con datos sobre precipitación pluvial.

Primero declaramos la matriz:

```
private int[][] datosLluvia =
 {{10, 7, 3, 28, 5, 6, 3},
 {12, 3, 5, 7, 14, 5, 8},
 {8, 5, 2, 1, 1, 4, 7}};
```

Para mostrar todos los datos:

```
private void mostrar() {
 datos.setText("");
 datos.setTabSize(3);
 String nuevaLinea = "\r\n";
 String tab = "\t";

 for (int ubicación = 0; ubicación <= 2; ubicación++) {
 for (int númeroDía = 0; númeroDía <= 6; númeroDía++) {
 datos.append(Integer.toString
 (datosLluvia[ubicación][númeroDía])
 + tab);
 }
 datos.append(nuevaLinea);
 }
}
```

El ciclo `for` interior recorre los distintos días, mientras que el ciclo `for` exterior recorre las distintas ubicaciones. Establecemos el tamaño del tabulador a un valor conveniente utilizando el método `setTabSize`.

Para cambiar un elemento en la matriz, debemos extraer el número de día, el número de ubicación y el nuevo valor de los datos de sus respectivos cuadros de texto:

```
private void cambiarValor() {
 int valorDatos;
 int númeroDía;
 int ubicación;

 númeroDía = Integer.parseInt(día.getText());
 ubicación = Integer.parseInt(lugar.getText());
 valorDatos = Integer.parseInt(nuevosDatos.getText());
 datosLluvia[ubicación][númeroDía] = valorDatos;

 mostrar();
 calcularTotal();
}
```

Para calcular el total de precipitación pluvial en todas las ubicaciones:

```
private void calcularTotal() {
 int total = 0;

 for (int ubicación = 0; ubicación <= 2; ubicación++) {
 for (int númeroDía = 0; númeroDía <= 6; númeroDía++) {
 total = total + datosLluvia[ubicación][númeroDía];
 }
 }
 cifras.setText("la precipitación total es " + Integer.toString(total));
}
```

Al ejecutar este programa tenga cuidado de introducir los números de fila en el rango de 0 a 2 y los números de columna en el rango de 0 a 6.

En este código podemos ver de nuevo que es muy común encontrar instrucciones `for` anidadas con matrices, ya que utilizan al máximo la uniformidad de las matrices.

## Fundamentos de programación

Una matriz es una colección de datos con un solo nombre (por ejemplo, `datosLluvia`). Una matriz se puede visualizar como una tabla bidimensional, con filas y columnas. Suponga que queremos representar los datos de precipitación pluvial para cada uno de los siete días de la semana en cada uno de tres lugares. Declaramos la siguiente matriz:

```
int[][] datosLluvia = new int[7][3];
```

Para acceder a los elementos de dicha matriz especificamos dos índices, que deben ser enteros; por ejemplo, `datosLluvia[4][2]`. Podemos considerar que el primer índice describe el número de fila y el segundo el número de columna. Al crear la matriz se especifican los tamaños de sus dos dimensiones: siete y tres en este ejemplo. Esta matriz tiene siete filas y tres columnas. Los índices siempre empiezan en 0. En este ejemplo los índices de las filas van de 0 a 6 y los índices de las columnas van de 0 a 2.

Los elementos de una matriz pueden ser de cualquier tipo: `int`, `double`, `String` o cualquier otro objeto. Pero todos los elementos de una matriz deben ser del mismo tipo: `int` en este ejemplo. La excepción es cuando se declara una matriz que consiste en objetos tipo `Object`. En este caso, dicha matriz puede dar cabida a cualquier mezcla de objetos.

Es común utilizar ciclos `for` anidados junto con las matrices.

En este libro exploramos tanto los arreglos unidimensionales como los bidimensionales. Java puede trabajar con matrices de hasta 60 dimensiones, pero es muy raro en la práctica utilizar más de 3.

## Errores comunes de programación

Un error común en Java es confundir la longitud de una matriz con el rango de índices válidos. Por ejemplo, la siguiente matriz:

```
int[][] tabla = new int[11][6];
```

tiene 11 filas y 6 columnas. El rango válido de índices para las filas es de 0 a 10. El rango válido de índices para las columnas es de 0 a 5. La referencia a `tabla[11][6]` hará que el programa se detenga y aparezca un mensaje de error.

## Resumen

- Un arreglo bidimensional o matriz es una colección de datos en una tabla, con filas y columnas. El programador debe asignar el nombre a una matriz.
- Una matriz se declara, al igual que otras variables, de la siguiente forma:

```
int[][] alices = new int[25][30];
```

en donde 25 es el número de filas y 30 el número de columnas.

- Para hacer referencia a un elemento individual de una matriz utilizamos índices enteros, como en el siguiente ejemplo:

```
alices[12][3] = 45;
```

## Ejercicios

### Operaciones básicas con matrices

**14.1 Manejador de datos** Escriba un programa que utilice una matriz de números de longitud  $4 \times 7$ , similar al programa de precipitación pluvial (con los resultados que se muestran en la figura 14.2). Extienda el programa para que realice las siguientes operaciones:

- Al oprimir un botón marcado como “sumas”, debe sumar los valores de cada una de las siete columnas y sumar todos los valores de cada una de las cuatro filas, además de mostrar los resultados en pantalla.
- Al oprimir un botón marcado como “mayor”, debe encontrar el valor más grande en cada fila, el valor más grande en cada columna y el valor más grande en toda la matriz.
- Al oprimir un botón marcado como “escalar”, debe multiplicar cada número en la matriz por un número introducido en un cuadro de texto (esto se podría utilizar para convertir de centímetros a pulgadas).

### Medidas estadísticas

**14.2** Extienda el programa de precipitación pluvial de manera que proporcione un botón para calcular el promedio diario de precipitación pluvial para cada ubicación. Por ejemplo, la precipitación pluvial promedio diaria en la ubicación 2 podría ser de 23.

Extienda aún más el programa para que tenga un botón que calcule la media y la desviación estándar de la precipitación pluvial diaria en cualquier ubicación. Por ejemplo, la media de la precipitación pluvial en cualquier ubicación podría ser de 19, con una desviación estándar de 6.4.

### Gráficos de barras y de pastel

**14.3** Extienda el programa de precipitación pluvial de manera que el usuario pueda especificar una fila (ubicación). Después tiene que mostrar la información como un gráfico de barras.

Extienda el programa para que muestre la información de una sola fila o columna como gráfico de pastel. Para ello utilice el método de biblioteca `fillArc` en la clase `Graphics`. Cree una matriz que contenga los colores disponibles (`Color.black`, `Color.lightGray`, `Color.blue`, etc.) y úsela para colorear las distintas rebanadas del pastel.

### Operaciones matemáticas

**14.4 Transpuesta** La transpuesta de una matriz es el término técnico que se utiliza para describir la acción de intercambiar los elementos de una matriz a través de una de las diagonales. Los números en la diagonal no cambian. Por ejemplo, si tenemos una matriz como se muestra en la figura 14.3, su transpuesta queda como se aprecia en la figura 14.4.

|    |    |    |    |
|----|----|----|----|
| 1  | 2  | 3  | 4  |
| 6  | 7  | 8  | 9  |
| 10 | 11 | 12 | 13 |
| 14 | 15 | 16 | 17 |

**Figura 14.3** La matriz antes de la transposición.

|   |   |    |    |
|---|---|----|----|
| 1 | 6 | 10 | 14 |
| 2 | 7 | 11 | 15 |
| 3 | 8 | 12 | 16 |
| 4 | 9 | 13 | 17 |

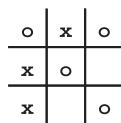
**Figura 14.4** La matriz transpuesta.

Escriba un programa que reciba como entrada los elementos de una matriz en forma similar al programa de precipitación pluvial. Deberá transponer la matriz al hacer clic en un botón y mostrar el resultado en pantalla.

## Juegos

**14.5 Tres en raya o gato** El gato (o ceros y cruces) se juega en una cuadrícula de  $3 \times 3$ , que en un principio está vacía. Hay dos jugadores y cada uno toma un turno. Uno de ellos coloca una cruz en un cuadro vacío. Después el otro coloca un cero en un cuadro vacío.

El ganador es la persona que obtenga una línea de tres ceros o tres cruces. Por ejemplo, la figura 14.5 representa una victoria para los ceros:



**Figura 14.5** Tres en raya o gato.

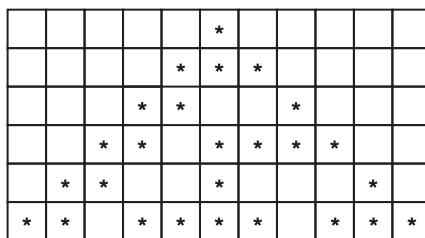
Los juegos pueden terminar en empate si ninguno de los jugadores logra obtener una línea.

Escriba un programa para jugar al gato. Debe haber sólo un botón para iniciar un nuevo juego. El programa debe mostrar los ceros y cruces en forma gráfica, cada uno en su propio panel. Para especificar un movimiento, el jugador humano debe hacer clic con el ratón en el panel en donde se vaya a colocar la cruz. El otro jugador será la computadora, que decidirá en dónde colocar los ceros al azar.

## Vida artificial

**14.6 Vida celular** Un organismo consiste en células individuales que están encendidas (vivas) o apagadas (muertas). Cada generación de vida consiste en una sola fila de células. Cada generación de vida (cada fila) del organismo depende de la anterior (al igual que en la vida real). El tiempo se desplaza de arriba hacia abajo. Cada fila representa una generación. Las vidas se ven como en la figura 14.6.

Al principio sólo hay una célula viva. El que una célula esté viva o muerta depende de una combinación de factores: si estaba o no viva en la última generación y si sus vecinas inmediatas estaban o no vivas en la última generación. Podemos ver que, incluso después de sólo cinco generaciones, está surgiendo un patrón. Estos patrones son muy sencillos e imitan a los patrones que se encuentran en los organismos vivos de la vida real. Las reglas son las siguientes:

**Figura 14.6** Vida celular.

Una célula sólo vive si:

- Estaba muerta, pero sólo su vecina izquierda estaba viva.
- Estaba muerta, pero sólo su vecina derecha estaba viva.
- Estaba viva, pero sus vecinas inmediatas estaban muertas.
- Estaba viva y sólo su vecina derecha estaba viva.

Por ejemplo, dada la generación que se muestra en la figura 14.7:

**Figura 14.7** Antes de un ciclo

- La primera célula vive, ya que aunque estaba muerta, su vecina derecha inmediata estaba viva.
- La segunda célula vive, ya que sólo su vecina derecha inmediata estaba viva.
- La tercera célula viva muere (¡suponemos que debido a la sobre población!).
- La cuarta célula muere.
- La quinta célula vive ya que, aunque estaba muerta, su vecina izquierda inmediata estaba viva.

Por lo tanto, la nueva generación es como se muestra en la figura 14.8:

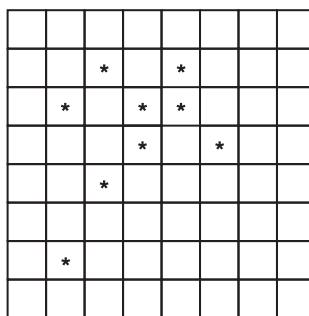
**Figura 14.8** Despues de un ciclo.

Escriba un programa que utilice una matriz para graficar el progreso de la forma de vida. Muestre el desarrollo en la pantalla en forma de asteriscos, como en las figuras anteriores. Agregue un botón que permita al usuario avanzar a la siguiente generación.

**14.7 El Juego de la vida de Conway** En esta forma de vida, un organismo también consiste en células individuales que están encendidas (vivas) o apagadas (muertas). Los organismos existen en un mundo de una cuadrícula bidimensional, como en la figura 14.9.

Las reglas que gobiernan este organismo son:

1. Si una célula viva tiene dos o tres vecinas, sobrevivirá. En caso contrario morirá por aislamiento o sobre población.



**Figura 14.9** Juego de la vida.

2. Si una célula vacía está rodeada por tres células exactamente, entonces nacerá una nueva célula viva para llenar el espacio.
3. Todos los nacimientos y muertes ocurren de manera simultánea.

Escriba un programa para simular este tipo de vida. Al principio el programa deberá permitir al usuario hacer clic en las células que vayan a estar vivas. Agregue un botón que permita al usuario avanzar a la siguiente generación y mostrarla en pantalla.

El programa necesita dos matrices: una para representar el estado actual de vida y otra para representar la siguiente generación. Después de crear cada nueva generación se intercambiarán las funciones de las dos matrices.

## Respuestas a las prácticas de autoevaluación

- 14.1** La columna 5 es sábado. Se vendieron 31 computadoras en la tienda 3 el jueves. Ésta es la fila 3 y columna 3.

**14.2**

```
String[][] tableroAjedrez = new String[8][8];
```

**14.3**

```
String[][] tableroAjedrez = new String[8][8];
for (int fila = 0; fila <= 7; fila++) {
 for (int col = 0; col <= 7; col++) {
 tableroAjedrez[fila][col] = "vacío";
 }
}
```

- 14.4** El valor es 8.

**14.5**

```
String[][] números =
{
 {"uno", "dos", "tres"},
 {"cuatro", "cinco", "seis"},
 {"siete", "ocho", "nueve"}
};
```

# CAPÍTULO **15**



## Manipulación de cadenas de texto

En este capítulo conoceremos:

- Las herramientas para trabajar con cadenas de texto.
- Los principales métodos de la clase **String**.

### ● Introducción

Las cadenas de caracteres son muy importantes en el software. Todos los lenguajes de programación tienen herramientas para la manipulación primitiva de caracteres, pero Java cuenta con una colección especialmente útil de métodos. En este capítulo reuniremos las herramientas para trabajar con cadenas que hemos utilizado hasta este momento y ampliaremos este conocimiento al estudiar el conjunto de métodos para procesamiento de cadenas.

He aquí algunas situaciones en las que se utilizan cadenas de texto:

- Para mostrar mensajes en la pantalla, lo cual podría requerir la colocación de texto en etiquetas.
- Para recibir texto del usuario. A menudo esto se hace mediante un campo de texto.
- Para almacenar datos en archivos. Cuando trabajemos con archivos en el capítulo 17 veremos que el contenido de muchos tipos de archivos se puede considerar como secuencias de cadenas. Además, los nombres de los archivos y las carpetas son cadenas de texto.
- Para buscar en páginas Web.
- Para guardar texto en memoria para los procesadores y editores de texto.

## Uso de cadenas de texto — un recordatorio

En esta sección hablaremos sobre las herramientas para trabajar con cadenas de texto que hemos visto hasta ahora.

Podemos declarar variables y proveer un valor inicial, como en el siguiente ejemplo:

```
String x, y;
String miCiudad;
String miNombre = "Parr";
String miPaís = new String("Japón");
```

En la última línea hemos mostrado la forma completa de proveer un valor inicial, ya que demuestra cómo, tras bambalinas, el uso de `new` siempre está involucrado en la asignación de espacio para el valor de la cadena de texto. Sin embargo, no hay ventaja al usar esta forma; se ejecuta una instrucción `new` sin importar que la solicitemos o no de manera explícita.

Podemos asignar una cadena a otra, como en:

```
x = "Inglaterra";
x = "Francia";
y = x;
x = ""; // una cadena de longitud cero
```

Esto muestra que la longitud de una cadena puede variar. En sentido estricto, la cadena anterior se destruye y es sustituida por un valor completamente nuevo. El espacio que ocupaba la cadena anterior estará disponible más adelante, mediante la recolección de basura, para que otras variables lo utilicen.

Podemos usar el operador `+` para concatenar cadenas de texto, como en el conocido ejemplo:

```
int número = 123;
campoTexto.setText("el valor es " + número);
```

Hay que tener cuidado al usar varios operadores `+`, como en:

```
campoTexto.setText("el valor es "+22+33);
```

La regla de Java establece que cuando un elemento es una cadena de texto los demás se convierten en cadenas, por lo que la instrucción anterior da como resultado lo siguiente:

```
el valor es 2233
```

Podemos forzar la realización de la suma numérica si hacemos lo siguiente:

```
campoTexto.setText("el valor es "+(22+33));
```

Cuyo resultado será:

```
el valor es 55
```

Una característica común del procesamiento de cadenas es empezar con una cadena vacía y agregarle elementos a medida que se ejecuta el programa. Podríamos utilizar la siguiente instrucción:

```
x = x + "algo";
```

que agrega texto al final de `x`. A esto se le conoce como “adjuntar”.

Podemos comparar cadenas, como en:

```
if (x.equals(y))...
```

Podemos crear arreglos de cadenas de texto (con índices que empiezan en 0), como en el siguiente ejemplo:

```
String [] ciudades = new String[10]; // 10 elementos, de 0 a 9
```

y podemos manipular elementos, como en:

```
ciudades[1] = "Los Ángeles";
```

Podemos convertir cadenas de texto a números **int** y **double**. Esto es útil cuando recibimos cadenas de un campo de texto (o de un archivo, como veremos más adelante). Por ejemplo, podríamos utilizar el siguiente código:

```
int valorEntero = Integer.parseInt(cadenaDatos);
```

Si la cadena de entrada no contiene un número válido, el método produce una indicación de error: una excepción. Estudiaremos la introducción de datos con más detalle posteriormente en el capítulo, y en el capítulo 16 veremos cómo se pueden detectar y controlar los errores.

Esto es lo que hemos visto hasta este momento. Ahora veremos los detalles sobre las cadenas de texto y los métodos disponibles..

## ● Los caracteres dentro de cadenas de texto

Los valores de cadena se colocan entre caracteres de doble comilla; pero, ¿qué pasa si necesitamos mostrar una doble comilla en pantalla? Podríamos intentar lo siguiente:

```
campoTexto.setText("¡Un "problema" engañoso!"); // incorrecto
```

Esta instrucción no se compilará, ya que la segunda `"` se considera como el final de la cadena completa. Lo que podemos hacer es usar el carácter de barra diagonal inversa `\`, el cual instruye a Java para que considere el siguiente carácter como un carácter ordinario, en vez de uno con significado especial. En la jerga, el carácter `\` se conoce como de “escape”. Así, la solución para nuestro problema de las comillas es:

```
campoTexto.setText("¡Un \"problema\" engañoso!");
```

En la pantalla aparecerá lo siguiente:

```
;Un "problema" engañoso!
```

Las secuencias `\n` y `\t` también tienen un significado especial en Java:

- `\n` representa el carácter de nueva línea.
- `\t` representa el carácter tabulador. Puede ser útil para alinear columnas de datos, como vimos con los arreglos en el capítulo 13.

## ● Una observación sobre el tipo `char`

En este libro es más importante la claridad de un programa que su velocidad de ejecución. Sin embargo, algunas veces nos encontramos con situaciones en las que podría ser conveniente evitar el

castigo en tiempo adicional por usar cadenas de texto. Como ejemplo considere el caso en que nuestras cadenas sólo contienen un carácter. Java cuenta con un tipo primitivo adicional llamado **char**, el cual almacena cada elemento como carácter Unicode de 16 bits, y como resultado de ello podemos realizar comparaciones entre valores **char** con la misma rapidez que con valores **int**, mientras que la comparación de cadenas requiere más tiempo. He aquí algunos ejemplos de su uso, en los cuales podemos declarar variables y asignarles un valor inicial:

```
char inicial = 'M';
char marcador = '\n';
char letra;
```

Observe el uso de los caracteres de comilla sencilla. Estas comillas sólo pueden contener un carácter. Aunque parezca que `\n` consta de dos caracteres, se reemplaza en tiempo de compilación por el carácter individual de nueva línea (o “intro”).

## ● La clase **String**

Como vimos en los ejemplos anteriores, podemos crear cadenas y arreglos de cadenas. También podemos concatenarlas (unirlas), mostrarlas en pantalla e introducirlas mediante un campo de texto. Lo que no hemos visto hasta ahora es el uso detallado de los métodos de biblioteca en la clase **String**. Al decir “cadenas” nos referimos a las instancias de la clase **String**.

## ● Los métodos de la clase **String**

Antes que nada, veamos algunos puntos generales. Cada carácter en una cadena tiene un índice (o número de posición). El primer carácter se enumera como **0**. Además, en muchos de los métodos de **String** hay que proveer los valores de los índices, y si por accidente proveemos un índice negativo o uno demasiado grande, se producirá un mensaje que contiene el nombre de excepción **StringIndexOutOfBoundsException**.

Aquí analizaremos los métodos más útiles de **String**. Como tal vez quiera ejecutar programas para confirmar su comprensión de cada método, le proveeremos un programa básico con dos campos de texto de entrada, un botón “hacerlo” y un campo de texto de salida. Al hacer clic en el botón se ejecutará la operación de cadenas que haya seleccionado. La figura 15.1 muestra una pantalla de ejemplo.

Para mostrar los resultados utilizamos un campo de texto que puede desplegar texto mediante el método **setText**.

El programa se puede utilizar con los métodos que mostraremos a continuación, los cuales proveen un ejemplo explícito. Dé un vistazo al siguiente código para saber en dónde debe insertar sus instrucciones. He aquí el programa:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class PlantillaString extends JFrame
 implements ActionListener {

 private JTextField campoCadena1, campoCadena2, campoResultado;
 private JLabel etiquetaCadena1, etiquetaCadena2, etiquetaResultado;
 private JButton botónIr;

 public static void main(String[] args) {
 PlantillaString marco = new PlantillaString();
 marco.setSize(250, 250);
 marco.crearGUI();
 marco.setVisible(true);
 }

 private void crearGUI() {
 setDefaultCloseOperation(EXIT_ON_CLOSE);
 Container ventana = getContentPane();
 ventana.setLayout(new FlowLayout());

 etiquetaCadena1 = new JLabel("Escriba la cadena1: ");
 ventana.add(etiquetaCadena1);

 campoCadena1 = new JTextField(20);
 ventana.add(campoCadena1);

 etiquetaCadena2 = new JLabel("Escriba la cadena2: ");
 ventana.add(etiquetaCadena2);

 campoCadena2 = new JTextField(20);
 ventana.add(campoCadena2);

 etiquetaResultado = new JLabel("el resultado es: ");
 ventana.add(etiquetaResultado);

 campoResultado = new JTextField(20);
 ventana.add(campoResultado);

 botónIr = new JButton("hacerlo");
 ventana.add(botónIr);
 botónIr.addActionListener(this);
 }

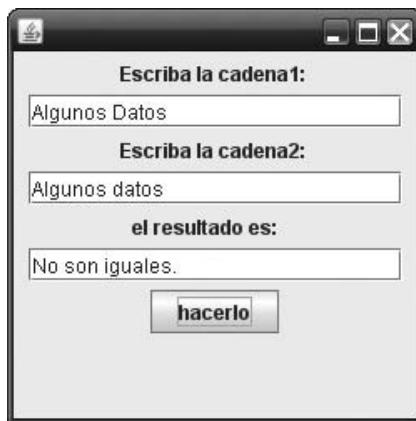
 public void actionPerformed(ActionEvent event) {
 String resultado ="";
 if (event.getSource() == botónIr) {
 String cadena1 = campoCadena1.getText();
 String cadena2 = campoCadena2.getText();
 // ejemplo de cadenas - equals
 //.... inserte aquí su código
 if(cadena1.equals(cadena2))
 resultado="Son iguales.";
 }
 }
}
```

```

 else
 resultado="No son iguales.";
 // fin del ejemplo

 campoResultado.setText(resultado);
 }
}
}

```



**Figura 15.1** El programa `PlantillaString`. Aquí se muestra el método `equals`.

## Comparación de cadenas

Los métodos más importantes en esta área son `equals`, `equalsIgnoreCase` y `compareTo`.

### `equals`

Se utiliza para comparar valores; no debemos usar `==` para esto. He aquí un ejemplo que se puede insertar en el programa de demostración:

```

// ejemplo de cadenas - equals
if(cadena1.equals(cadena2))
 resultado="Son iguales.";
else
 resultado="No son iguales.";
// fin del ejemplo

```

### equalsIgnoreCase

Provee una herramienta similar a `equals`, excepto que se ignora el uso de mayúsculas y minúsculas en cada carácter. “**E**sta Cadena” se considerará igual a “**e**sta **c**adena”. Para experimentar con esto use el código anterior, sólo sustituya la llamada a `equals` por una llamada a `equalsIgnoreCase`.

### compareTo

Imagine que tenemos varias cadenas (que tal vez contengan nombres de personas) que necesitamos colocar en orden alfabético. El método `compareTo` nos permite hacer esto. Tras bambalinas se utilizan los códigos enteros internos de los caracteres. El único punto a considerar es que las letras minúsculas tienen códigos más altos que las letras mayúsculas. He aquí algunos ejemplos:

```
ant va antes que bee
and va antes que ant
an va antes que and
ANT va antes que BEE
INSECT va antes que ant
Insect va antes que ant
INSECT va antes que insect
```

El método `compareTo` devuelve un resultado entero con el siguiente significado:

- 0 si las cadenas son iguales.
- Un valor negativo si el objeto cadena va antes que el parámetro.
- Un valor positivo si el objeto cadena va después que el parámetro.

Si utilizamos:

```
n = "ant".compareTo("bee");
```

entonces `n` se establecerá en un valor negativo. Podemos comprobar este resultado con el siguiente código:

```
//ejemplo de cadenas - compareTo
int n = cadena1.compareTo(cadena2);
if (n == 0)
 resultado = "son iguales";
else if (n < 0)
 resultado = "cadena1 va antes que cadena2";
else
 resultado = "cadena2 va antes que cadena1";
// fin del ejemplo
```

## ● Corrección de cadenas

En esta sección veremos los métodos para modificar una cadena de texto. Tras bambalinas, estos métodos crean una nueva cadena, en vez de modificar la original.

### **replace**

Este método reemplaza un carácter por otro, en toda la cadena. Por ejemplo:

```
cadena1= "Mississippi".replace('i', 'a');
```

colocaría **"Massassappa"** en **cadena1**. Puede experimentar con el siguiente código. Insértelo en el programa de demostración.

```
// ejemplo de cadenas - replace
resultado = cadena1.replace('a', 'A'); // reemplaza toda 'a' por 'A'
// fin del ejemplo
```

### **toLowerCase**

El método **toLowerCase** convierte cualquier cadena con letras mayúsculas en una cadena con letras minúsculas, como en el siguiente ejemplo:

```
cadena1 = "Versión 1.1";
resultado = cadena1.toLowerCase();
```

que coloca **"versión 1.1"** en **resultado**. Para experimentar con esto utilice las siguientes líneas de código:

```
// ejemplo de cadenas - toLowerCase
resultado = cadena1.toLowerCase();
// fin del ejemplo
```

### **toUpperCase**

El método **toUpperCase** realiza una operación similar a la de **toLowerCase**, sólo que cambia las letras minúsculas a sus equivalentes en mayúsculas. Por ejemplo:

```
cadena1 = "Java";
resultado = cadena1.toUpperCase();
```

### **trim**

El método **trim** elimina espacios en blanco de ambos extremos de una cadena. “Espacio en blanco” no sólo significa caracteres de espacio, sino también de nueva línea y de tabulación. Si utilizamos:

```
cadena1 = " Centro ";
resultado = cadena1.trim();
```

entonces **resultado** se convertirá en “Centro”. El código de ejemplo que puede utilizar es:

```
// ejemplo de cadenas - trim
resultado = cadena1.trim();
// fin del ejemplo
```

## Análisis de cadenas

Los siguientes métodos nos permiten analizar una cadena de texto; por ejemplo, extraer una sección de la misma. Con frecuencia, a la sección de una cadena se le llama subcadena.

### **length**

El método **length** proporciona el número de caracteres en una cadena de texto, como en:

```
int n = "Programación en Java".length;
```

Aquí, **n** se establece en **20**. Si coloca el siguiente código dentro del programa de plantilla se mostrará en pantalla la longitud de una cadena que introduzca:

```
// ejemplo de cadenas - length
resultado = "la longitud es " + cadena1.length();
// fin del ejemplo
```

El proceso para obtener la longitud de un arreglo es distinto. Por ejemplo, la longitud de un arreglo llamado **tabla** se obtiene mediante **tabla.length**.

### **substring**

El método **substring** extrae una parte específica de una cadena. La llamada provee la posición inicial y la posición **1** lugar más allá que el último carácter a extraer. ¡Tenga cuidado con este segundo parámetro! En el siguiente extracto:

```
cadena1 = "posición";
resultado = cadena1.substring(2,5);
```

a **resultado** se le asigna la cadena “sic”. La primera posición en una cadena se enumera como **0** y el último carácter siempre está en **length() - 1**, como podemos ver en la siguiente tabla:

|          |   |   |   |   |   |   |   |   |
|----------|---|---|---|---|---|---|---|---|
| Índice   | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| Carácter | p | o | s | i | c | i | ó | n |

(La longitud es 8)

He aquí el código para el programa de ejemplo, que muestra la entrada sin el primer y último caracteres.

```
// ejemplo de cadenas - substring
resultado = cadena1.substring(1, cadena1.length()-1);
fin del ejemplo
```

## PRÁCTICA DE AUTOEVALUACIÓN

15.1 Explique el efecto del siguiente código:

```
String palabra = "posición";
String s = palabra.substring(2, palabra.length());
```

### charAt

El método `charAt` devuelve el carácter en una posición especificada. Cabe mencionar que el resultado es de tipo `char`, no una cadena de longitud 1. En algunas situaciones esto puede ser más rápido que el uso de `substring`. He aquí un ejemplo:

```
char c1,c2;
cadena1 = "posición";
c1 = cadena1.charAt(1); // c1 se convierte en "o"
c2 = cadena1.charAt(4); // se convierte en "c"
```

### indexOf

Este método determina si una subcadena está dentro de una cadena. Podemos proveer un desplazamiento para especificar en dónde debe empezar la búsqueda. Por ejemplo:

```
int n = "mississippi".indexOf("is",4);
```

significa que busca "is" a partir de la posición 4. Esto hace que `n` reciba un 4, que muestra la posición del segundo "is" (recuerde que la primera posición de una cadena se enumera como 0).

Pero si usamos lo siguiente:

```
int n = "mississippi".indexOf("is",5);
```

entonces `n` se convierte en -1 para indicar que no se encontró la cadena (el valor 5 indica una búsqueda a partir de la tercera "s"). He aquí código para el programa de ejemplo, el cual informa si una cadena contiene una subcadena:

```
//ejemplo de cadenas - indexOf
if (cadena1.indexOf(cadena2,0) >= 0)
 resultado = cadena2 + " existe dentro de " + cadena1;
else
 resultado = cadena2 + " no existe dentro de " + cadena1;
//fin del ejemplo
```

### **lastIndexOf**

Este método es similar en concepto a `indexOf`, sólo que devuelve la posición de la ocurrencia de más a la derecha de una subcadena. Se devuelve el valor `-1` si no hay una coincidencia. He aquí un ejemplo:

```
int lugar = "//a.b.c/directorio/archivo".lastIndexOf("//");
```

Se devuelve el valor `18`.

### **endsWith**

Este método se utiliza para averiguar si una cadena termina con una subcadena específica. Aunque podemos utilizar una combinación de otros métodos para lograr esto, el método que se proporciona es menos propenso a errores. El método devuelve un valor `boolean`. Por ejemplo:

```
boolean r = "http://ruta//".endsWith("//");
```

establece `r` a `true`.

He aquí código para el programa de ejemplo, el cual determina si una subcadena está presente al final de otra cadena:

```
// ejemplo de cadenas - endsWith
if (cadena1.endsWith(cadena2))
 resultado = "cadena1 termina con " + cadena2;
else
 resultado = "cadena1 no termina con " + cadena2;
// fin del ejemplo
```

## **La clase StringTokenizer**

Hemos visto que se puede buscar una subcadena dentro de otras cadenas y, con base en el resultado, se pueden dividir en partes. Sin embargo, cuando nuestros datos contienen subcadenas repetidas separadas por caracteres especiales, la división se puede realizar con más facilidad mediante la clase `StringTokenizer`. He aquí algunas cadenas típicas que podría ser conveniente separar:

```
enero 21 5
4,6 ,7,10,10,12,13, 15, ,21,20,19, ,8
```

En ambas cadenas existe el concepto de un delimitador, el cual descompone los elementos separados (o tokens). En el primer ejemplo, que representa las horas de sol durante el **21 de enero**, el delimitador es un espacio o una serie de espacios. En el segundo ejemplo, el delimitador es una coma, pero también hay espacios presentes. Las cadenas anteriores son representativas del tipo de datos que existen en los archivos. Algunas veces estos archivos son creados por otros programas de Java, o se pueden haber producido al exportar datos de una hoja de cálculo (para detalles sobre el procesamiento de archivos, consulte el capítulo 17).

He aquí cómo podemos ver las cadenas, token por token:

```
String ejemplo1 = "enero 21 5";
String mes, día, horas;
StringTokenizer datosSol = new StringTokenizer(ejemplo1, " ");
mes = datosSol.nextToken();
día = datosSol.nextToken();
horas = datosSol.nextToken();
```

En el ejemplo anterior creamos una nueva instancia de la clase **StringTokenizer** llamada **datosSol**. En el constructor proveemos la cadena que se va a descomponer y una cadena que contiene el delimitador (o delimitadores). Para lidiar con la primera cadena utilizamos `" "`; el resultado que se produce es que cualquier espacio se considerará como un delimitador en vez de considerarlo como datos. Después utilizamos el método **nextToken** para obtener cada elemento en turno. Dependiendo del problema, podríamos optar por convertir las cadenas de dígitos en un valor **int**.

Para la segunda cadena vamos a suponer que no sabemos de antemano cuántos tokens contiene. Podemos usar el método **hasMoreTokens**, el cual devuelve verdadero si hay más datos para que **nextToken** los obtenga. También proveemos una cadena delimitadora que consiste en `" ,"` para indicar que los espacios y/o las comas son delimitadores. Podríamos usar el siguiente ejemplo:

```
String ejemplo2 = "4,6 ,7,10,10,12,13, 15, 21,20,19, 8";
String elemento;
StringTokenizer listaNúmeros = new StringTokenizer(ejemplo2, ", ");
while (listaNúmeros.hasMoreTokens()) {
 elemento = listaNúmeros.nextToken();
 // ... procesar elemento
}
```

Para resumir, los principales métodos de la clase **StringTokenizer** son:

```
nextToken()
hasMoreTokens()
```

La clase **StringTokenizer** está en la biblioteca **util**, por lo que debemos colocar:

```
import java.util.*;
```

en el encabezado de nuestros programas.

## Conversiones de cadenas

Los datos que mostramos en pantalla o que se introducen desde el teclado están en forma de cadenas; pero la forma interna que utilicemos puede ser distinta. Por ejemplo, tal vez tengamos que recibir como entrada una serie de dígitos como una cadena de texto, para después convertirlos en

un tipo **int**. Java cuenta con diversos métodos de “conversión” o “análisis” que convierten las cadenas en una variedad de tipos; además, muchas clases de la biblioteca de Java proveen un método **toString**, el cual produce una representación de cadena de una instancia.

Recuerde que **int** y **double** son tipos primitivos “integrados”, por lo cual no tienen métodos asociados. Para compensar esta situación, Java tiene clases que proveen herramientas para los tipos primitivos, que los envuelven de manera que parezcan instancias de clases. Las más relevantes son:

```
Integer
Double
Boolean
Character
```

Observe con cuidado que los nombres siguen la convención de Java de usar una letra mayúscula para empezar el nombre de la clase. Así, **double** es el tipo integrado y **Double** es la clase que lo envuelve. Necesitamos saber acerca de las clases anteriores ya que proveen métodos de “conversión” además de **toString** para usarlos en las conversiones. He aquí algunos ejemplos:

Para convertir un **int** a **String**:

```
int n = 123;
String s = Integer.toString(n); // s toma el valor de "123"
```

Observe que el nombre de la clase **Integer** se coloca antes que el nombre del método estático.

Para convertir un **double** a **String**:

```
double d = 12.34;
String s = Double.toString(d); // s toma el valor de "12.34"
```

Observe de nuevo que el nombre de la clase **Double** (con D mayúscula) se coloca antes que el nombre del método estático.

Para convertir un **String** a **int**:

```
String s = "1234";
int n = Integer.parseInt(s);
```

El método **parseInt** devuelve un tipo **int**. Un uso común de esto es cuando se obtiene una cadena de dígitos de un campo de texto, como en el siguiente ejemplo:

```
n = Integer.parseInt(unCampoTexto.getText());
```

Para convertir un **String** a **double** podemos usar **parseDouble**, como en el siguiente ejemplo:

```
String s = "12.34";
double d = Double.parseDouble(s);
```

Al convertir la entrada del usuario en números siempre existe el problema de los errores. El usuario podría escribir:

```
123XY3
```

Aquí hemos hecho la suposición de que un error en la entrada no es peligroso y de hecho, nuestros programas que usan la conversión anterior simplemente no podrán lidiar con los errores. Esto

no es aceptable para el software serio; en el capítulo 16 veremos cómo detectar y manejar esos errores.

## ● Parámetros de cadena

Como vimos en nuestro estudio de los métodos disponibles, las cadenas se pueden pasar como parámetros y se pueden devolver como resultados; su uso es razonablemente intuitivo.

He aquí un ejemplo. Vamos a crear un método que duplica una cadena; así, “**hola**” se convierte en “**holahola**”.

```
private String duplicarCadena(String cualquiera) {
 return cualquiera + cualquiera;
}
```

Ésta es la forma en que podríamos llamar al método:

```
String s1 = "hola";
s1 = duplicarCadena(s1);
```

Pasamos la cadena al método en forma de parámetro. Se crea una nueva cadena y se devuelve como valor de retorno. Finalmente, este nuevo valor se asigna a **s1**.

## ● Un ejemplo de procesamiento de cadenas

En esta sección veremos la creación de un método de procesamiento de cadenas que realiza una tarea común: examinar una cadena y sustituir cada ocurrencia de una subcadena específica por otra subcadena (de una longitud potencialmente distinta). Cabe mencionar que la clase **String** tiene un método **replace**, pero éste sólo puede manejar caracteres individuales. Vamos a crear un método que trabaja con subcadenas. He aquí un ejemplo. Si tenemos la siguiente cadena:

“ir a jugar o no ir a jugar”

y reemplazamos cada ocurrencia de “jugar” por “comer”, crearemos la siguiente cadena:

“ir a comer o no ir a comer”

El proceso básico es utilizar **indexOf** para determinar la posición de una subcadena; en este caso, “**ser**”. Después formamos una nueva cadena compuesta por la parte izquierda de la cadena, la parte derecha y la cadena de reemplazo al centro. Tenemos entonces lo siguiente:

“ir a” + “comer” + “ no ir a jugar”

El proceso se debe repetir hasta que no haya más ocurrencias de “jugar”. Hay dos casos problema:

- El usuario de **replace** nos pide reemplazar un valor de “”. ¡Podríamos considerar que antes de cualquier cadena hay un número infinito de esas cadenas vacías! Nuestra forma de proceder en este caso es simplemente devolver la cadena original sin cambios.

- La cadena de reemplazo contiene la cadena que se va a reemplazar. Por ejemplo, podríamos tratar de cambiar “**ser**” por “**serpiente**”. Para evitar que se realice un número infinito de reemplazos, nos aseguramos de que sólo se consideren subcadenas en la parte derecha de la cadena. Utilizamos la variable **iniciarBúsqueda** para saber en dónde está el inicio de la parte derecha de la cadena.

El código completo es:

```
private String replace(String original, String de,
 String para) {
 String parteIzq, parteDer;
 int iniciarBúsqueda = 0;
 int lugar = original.indexOf(de);
 if (de.length() != 0) {
 while (lugar >= iniciarBúsqueda) {
 parteIzq = original.substring(0, lugar);
 parteDer = original.substring(lugar +
 de.length(), original.length());
 original = parteIzq + para + parteDer;
 iniciarBúsqueda = parteIzq.length() + para.length();
 lugar = original.indexOf(de);
 }
 }
 return original;
}
```

Ésta es la forma en que podríamos llamar a nuestro método:

```
String original = "ir a jugar o no ir a jugar";
String modificada = replace(original, "jugar", "comer");
```

En definitiva el extracto anterior no trabajará en forma aislada, por lo que ahora lo vamos a incorporar a un programa.

### Ejemplo práctico: Frasier

En 1970, Joseph Weizenbaum escribió un programa conocido como ELIZA para simular un estilo particular de psiquiatra. Era un programa simple en cuanto a que se esforzaba muy poco por comprender el sentido de lo que escribían los usuarios (pacientes). Por ejemplo, si el paciente escribía:

```
Yo estoy triste
```

entonces ELIZA podría responder con lo siguiente:

```
usted está triste - ¿por qué?
```

De manera similar, si el paciente escribía:

```
Yo estoy Java
```

entonces ELIZA podría responder con:

```
Usted está Java - ¿por qué?
```

En esta sección presentaremos una versión aún más simplificada, a la cual llamaremos Frasier en honor al personaje de televisión estadounidense. La metodología del diseño será separar la interfaz de usuario del procesamiento de las cadenas. Vamos a crear una clase llamada **Psiquiatra** con dos métodos: uno para aceptar una pregunta y el otro para producir una respuesta. He aquí la clase:

```
import java.util.*;

public class Psiquiatra {

 private String pregunta;
 private String respuesta;
 private Random valorAleatorio = new Random();

 public void ponerPregunta(String p) {
 pregunta = " " + p + " ";
 }

 public String obtenerRespuesta() {
 int variación = valorAleatorio.nextInt(3);
 switch (variación) {
 case 0:
 respuesta = transformarPregunta();
 break;
 case 1:
 respuesta = "¿Por qué siente eso?";
 break;
 case 2:
 respuesta = "¡Por favor sea sincero!";
 break;
 }
 return respuesta;
 }

 private String transformarPregunta() {
 if (pregunta.indexOf(" Yo ") >= 0) {
 String respuestaTemp = replace(pregunta,
 " Yo ", " usted ");
 ...
 }
 }
}
```

```
 respuestaTemp = replace(respuestaTemp,
 " estoy ", " está ");
 return replace(respuestaTemp, " mi "," su ") +
 "-¿por qué?";
}
else
 if (pregunta.indexOf(" no ") >= 0)
 return "¿no? - ¡qué negativo! Por favor explique...";
 else
 return "\\" + pregunta + "\\\"-Por favor explique...";
}

private String replace(String original, String de,
 String para) {
 String parteIzq, parteDer;
 int iniciarBúsqueda = 0;
 int lugar = original.indexOf(de);
 if (de.length() != 0) {
 while (lugar >= iniciarBúsqueda) {
 parteIzq = original.substring(0, lugar);
 parteDer = original.substring(lugar +
 de.length(), original.length());
 original = parteIzq + para + parteDer;
 iniciarBúsqueda = parteIzq.length() + para.length();
 lugar = original.indexOf(de);
 }
 }
 return original;
}
}
```

La interfaz de usuario es simple y directa. Consiste en dos campos de texto, uno para aceptar la entrada del usuario y otro (no editable) para mostrar la salida del programa. La figura 15.2 muestra una pantalla de ejemplo y he aquí la clase de la interfaz de usuario:

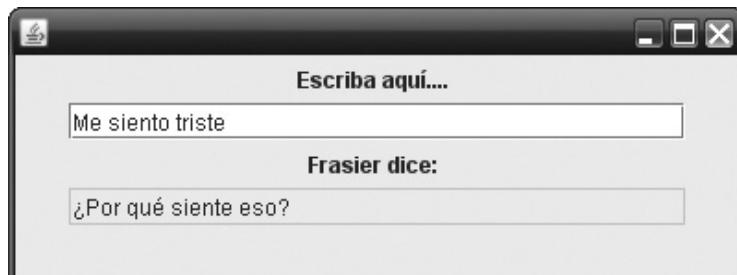


Figura 15.2 El programa de Frasier.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class PregunteAFrasier extends JFrame
 implements ActionListener {

 private JTextField campoPregunta, campoRespuesta;
 private JLabel etiquetaPsiquiatra, etiquetaPregunta;
 private Psiquiatra frasier;

 public static void main(String[] args) {
 PregunteAFrasier marco = new PregunteAFrasier();
 marco.setSize(400, 150);
 marco.crearGUI();
 marco.setVisible(true);
 }

 private void crearGUI() {
 setDefaultCloseOperation(EXIT_ON_CLOSE);
 Container ventana = getContentPane();
 ventana.setLayout(new FlowLayout());

 etiquetaPregunta = new JLabel("Escriba aquí.... ");
 ventana.add(etiquetaPregunta);

 campoPregunta = new JTextField(30);
 ventana.add(campoPregunta);
 campoPregunta.addActionListener(this);

 etiquetaPsiquiatra = new JLabel("Frasier dice: ");
 ventana.add(etiquetaPsiquiatra);

 campoRespuesta = new JTextField(
 "Continúe, por favor... Estoy escuchando.", 30);
 campoRespuesta.setEditable(false);
 ventana.add(campoRespuesta);

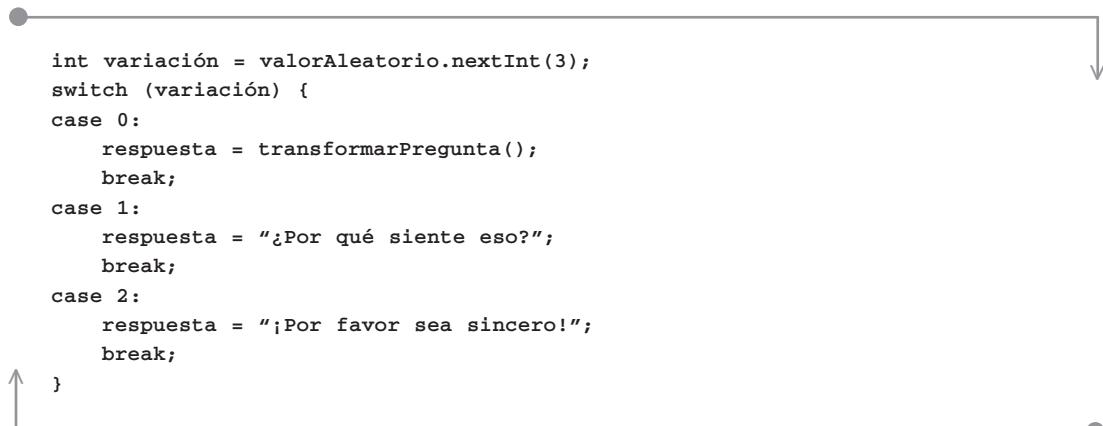
 frasier = new Psiquiatra();
 }

 public void actionPerformed(ActionEvent event) {
 if (event.getSource() == campoPregunta) {
 String suValor = campoPregunta.getText();
 frasier.ponerPregunta(suValor);
 campoRespuesta.setText(frasier.obtenerRespuesta());
 }
 }
}
```

El manejo de eventos es simple y directo: cuando el usuario oprime “Intro”, la pregunta se pasa a Frasier y después se le pide una respuesta. Aquí es donde se lleva a cabo el verdadero trabajo con las cadenas:

```
if (event.getSource() == campoPregunta) {
 String suValor = campoPregunta.getText();
 frasier.ponerPregunta(suValor);
 campoRespuesta.setText(frasier.obtenerRespuesta());
}
```

Para que las respuestas parezcan más humanas, agregamos un elemento de azar a **Psiquiatra**:



```
int variación = valorAleatorio.nextInt(3);
switch (variación) {
 case 0:
 respuesta = transformarPregunta();
 break;
 case 1:
 respuesta = "¿Por qué siente eso?";
 break;
 case 2:
 respuesta = "¡Por favor sea sincero!";
 break;
}
```

El entero aleatorio provee tres casos. En dos de ellos producimos una respuesta estándar, pero en el otro caso transformamos la pregunta; por ejemplo, reemplazamos cada ocurrencia de “yo” por “usted”. Agregamos espacios adicionales al inicio y al final de la pregunta para ayudar a detectar palabras completas. Tenga en cuenta que el programa no conoce el significado o la gramática del español. Para agregar esto se requeriría un enorme esfuerzo de programación.

## Fundamentos de programación

- Una instancia de la clase **String** puede contener cualquier número de caracteres.
- Los métodos de la clase **String** pueden manipular cadenas de texto.

## Errores comunes de programación

- Las cadenas son objetos y la clase **String** proporciona métodos. El uso correcto es, por ejemplo:

```
int n = cadena1.length();
```

en vez de:

```
int n = length(cadena1);
```

- No use == para comparar cadenas. Use los métodos **equals** o **equalsIgnoreCase**, como en:

```
if (respuesta.equals("ir")) {
 // etc ...
```

- Para introducir elementos **int** y **double** se usan cadenas de texto, lo cual es complicado. Siga nuestros ejemplos con cuidado.
- Tenga cuidado con el parámetro final de **substring**. Este parámetro indica la posición un lugar más allá del elemento que se va a extraer.
- Un mensaje **StringIndexOutOfBoundsException** lo llevará a una línea de su programa en donde uno de los parámetros índices es negativo, o trata de acceder a un carácter que está más allá del final de la cadena.

## Secretos de codificación

Los métodos de la clase **String** requieren que les proporcionemos una cadena de texto sobre la que puedan operar, como en el siguiente ejemplo:

```
String s = "demo";
int n = s.length();
```

Cabe mencionar que podemos proveer una cadena literal o la llamada a un método que devuelva una cadena, como en el siguiente ejemplo:

```
n = "otro demo".length();
n = s.substring(0, 2).length();
```

## Nuevos elementos del lenguaje

- El tipo primitivo (integrado) **char**.
- El \ como carácter de escape.

## Resumen

- Las instancias de la clase **String** contienen una secuencia de caracteres. El primer carácter está en la posición 0.
- Se pueden declarar y crear instancias de **String**, por ejemplo:

```
String nombre = "una secuencia de valores char";
```

- Los métodos más útiles para manipulación de cadenas son:

- Comparar cadenas

```
equals
equalsIgnoreCase
compareTo
```

- Corregir cadenas

```
replace
toLowerCase
toUpperCase
trim
```

- Examinar cadenas

```
length
substring
charAt
indexOf
lastIndexOf
endsWith
 StringTokenizer (clase)
```

- Conversión

```
toString
parseInt
parseDouble
```

(junto con las clases envolventes **Double** e **Integer**).

## Ejercicios

- 15.1** Escriba un programa que reciba como entrada dos cadenas mediante campos de texto y que las una. Muestre la cadena resultante en un campo de texto.
- 15.2** Escriba un programa que reciba como entrada una cadena de texto y determine si es un palíndromo o no. Un palíndromo se lee igual al derecho que al revés; por ejemplo, “**abba**” es un palíndromo.

- 15.3** Escriba un programa que reciba como entrada una cadena que pueda ser un número `int` o `double`. Muestre el tipo del número. Suponga que un `double` contiene un punto decimal.
- 15.4** Modifique el programa de Frasier para hacerlo más humano; agregue más variación a las respuestas.
- 15.5** Escriba un programa que genere código de Java para una instancia de `JButton`. Proporcione dos campos de texto, en los que el usuario debe introducir el nombre del botón requerido y su leyenda (sin comillas). Un clic de botón debe iniciar el proceso de generación. Para la salida use dos áreas de texto:
- La primera debe mostrar las declaraciones `private` apropiadas.
  - La segunda debe mostrar el código que crea los botones, los agrega a la ventana y se registra para los eventos de acción.

Recuerde que el usuario puede copiar código de las áreas de texto y pegarlo en el programa que desea crear. Asegúrese de que se puedan especificar varios botones y que el código creado se adjunte al demás código en las áreas de texto.

- 15.6** Mejore el ejercicio anterior con una tercera área de texto que contenga el texto de un método `actionPerformed`. Debe contener una serie de instrucciones `if` para identificar cuál fue el botón en el que se hizo clic.
- 15.7** Escriba un programa que permita introducir cálculos en un campo de texto, de la siguiente forma:

```
123 + 45
6783 - 5
```

(es decir, dos enteros con el signo + o – entre ellos y espacios que separan los elementos); además, debe mostrar el resultado del cálculo en un segundo campo de texto.

- 15.8** Extienda el ejercicio 15.7 de manera que reciba datos de la siguiente forma:

```
12 + 345 - 44 - 23 - 57 + 2345
```

Suponga que el usuario no cometerá errores.

Sugerencia: el patrón de dicha entrada es un número inicial seguido de cualquier número de pares de operadores/números. Use  `StringTokenizer`. Su programa debe ser capaz de manejar el número inicial y después iterar para manejar los siguientes pares.

- 15.9** Extienda el ejercicio 15.7 de manera que pueda recibir datos de las siguientes dos maneras:

```
setm 2 426
12 + m2
```

La instrucción `setm` debe ir seguida de dos números. El primero hace referencia a una ubicación de memoria enumerada del 0 al 9, y el segundo es un número que se debe almacenar en la memoria. Ahora se pueden realizar cálculos utilizando enteros como en los ejercicios anteriores, y también nombres de memoria. El cálculo anterior produce como resultado 438 (sugerencia: utilice un arreglo `int` para representar la memoria). Extienda su programa de manera que se procesen las siguientes formas de entrada:

```
m3 = 12 + m5 - 328 - m7
mostrar m3
```

- 15.10** Escriba un programa que almacene enteros como cadenas, hasta una longitud de 50 dígitos. Debe permitir la entrada de dos cadenas de ese tipo; además, debe proveer botones para seleccionar suma o resta (sugerencia: recuerde cómo aprendió a sumar números con la mano). Proporcione botones para multiplicar y dividir; implemente estas características mediante la suma o resta repetidas.

## Respuesta a la práctica de autoevaluación

**15.1** El valor de **s** es **sición**.

# CAPÍTULO **16**



## Excepciones

En este capítulo conoceremos:

- Qué es una excepción.
- Por qué las excepciones son útiles.
- Las herramientas de manejo de excepciones de Java.

### ● Introducción

El término *excepción* se utiliza en Java para transmitir la idea de que algo ha salido mal: en términos comunes, que ha ocurrido un error. Es una “circunstancia excepcional”. Cabe mencionar que nos referimos a excepcional en el sentido de inusual, en vez de maravilloso. Seguramente habrá notado al usar las computadoras que hay una variedad de circunstancias en las que el software puede funcionar de manera errónea, pero el software de buena calidad debe hacer frente a los errores predecibles de una manera satisfactoria. Por ejemplo, a continuación le presentamos algunas situaciones extrañas que se presentan en un procesador de palabras, con los posibles resultados (algunas veces insatisfactorios):

- El sistema le pide que escriba un tamaño de fuente como número, pero usted escribe un nombre. El sistema podría terminar y regresar el control de la ejecución al sistema operativo, o podría ignorar lo que usted acaba de escribir y dejar el tamaño de fuente como estaba, o podría mostrar un mensaje de ayuda e invitarlo a intentar de nuevo.
- Usted trata de abrir un archivo que no se encuentra en el disco. Las respuestas podrían ser similares al caso anterior.
- Usted trata de imprimir un archivo pero su impresora se quedó sin papel. De nuevo, esto podría predecirse, y podemos escribir el software de manera que tome acciones adecuadas. Sin embargo, esto depende del hecho de que el software pueda acceder al estado actual de la impresora. En las

impresoras recientes el software puede examinar varios bits de estado que indican si se agotó el papel, si está conectada/desconectada, si hubo un atasco de papel, etc.

## PRÁCTICA DE AUTOEVALUACIÓN

- 16.1** En los casos anteriores, decida cuál sería el mejor curso de acción que el procesador de palabras debería tomar.

Ahora veamos por qué es necesario tener una forma de notificación de los errores y cómo podríamos proveerla.

Al construir sistemas de software y hardware, la mayoría de ellos existen como elementos preempaquetados, como los tableros de circuitos, o las clases y los métodos de Java. Para simplificar el proceso de diseño es imprescindible considerar que estos elementos están encapsulados; no queremos preocuparnos por su funcionamiento interno, pero es vital que los componentes que utilicemos proporcionen alguna indicación en caso de situaciones de error. Entonces podemos escribir el software de manera que detecte dicha notificación y tome una acción alternativa. Pero ¿qué acción debe tomar? ¡Ésta es la parte difícil! Los sistemas complejos constan de una jerarquía de métodos; algunas excepciones se pueden manejar de manera local en el método en el que ocurren, pero algunos casos más graves tal vez necesiten llevarse más arriba a los métodos de nivel superior. Todo depende de la naturaleza del error. En resumen, hay distintas categorías de errores que tal vez deban manejarse en distintos lugares.

He aquí una analogía que ilustra esto. Imagine una organización en la que el director general empieza por dar instrucciones a sus gerentes. A su vez, ellos podrían instruir a los programadores y técnicos. Pero las cosas pueden salir mal. He aquí dos casos:

- Una de las impresoras se queda sin papel. Por lo general, el técnico se hace cargo de ello. En el extraño caso de que la organización se quede sin papel, tal vez habría que informar a uno de los gerentes.
- Un técnico tropieza con un cable y se rompe una pierna. El director general se debe hacer cargo de las excepciones de esta categoría (que podrían provocar una acción legal, etc.).

La analogía es que cada persona que hace un trabajo es un método. El trabajo lo inició alguien superior a ellos. Cuando hay errores, en realidad necesita haber un plan que indique quién va a manejar un tipo específico de error. Las herramientas de manejo de excepciones de Java nos permiten hacer esto.

Como dijimos antes, las cosas pueden salir mal. Pero, ¿realmente necesitamos una herramienta especial para los errores? ¿No podríamos utilizar la instrucción `if`? Podríamos pensar en un código similar a éste:

```
if (algo sale mal)
 manejar el problema;
else
 manejar la situación normal;
```

Aquí hemos utilizado una mezcla de español y Java para establecer el punto principal. Pero si tenemos varias llamadas a métodos, de las cuales cualquiera podría producir un error, la lógica se vuelve compleja y podría embrollar el caso normal. La sencilla secuencia inicial de:

```
hacerA()
hacerB()
hacerC()
```

se convertiría en:

```
hacerA()
if (hacerA tuvo errores)
 manejar el problema de hacerA;
else
 hacerB();
 if (hacerB tuvo errores)
 manejar el problema de hacerB;
 else
 hacerC();
 if (hacerC tuvo errores)
 manejar el problema de hacerC;
 else
etc...
```

Los casos de error (que esperamos no ocurran muy seguido) dominan la lógica; esta complejidad puede hacer que los programadores no quieran utilizar este código. De hecho, las herramientas de Java para el manejo de excepciones nos permiten apegarnos a la codificación para el caso normal y manejar las excepciones en un área separada del programa.

El esquema anterior basado en instrucciones `if` tiene una desventaja adicional en Java. Como vimos, los métodos pueden tener parámetros de entrada y pueden devolver un solo resultado. ¿Qué pasa si un método ya devuelve un resultado como parte de su trabajo? No puede devolver con facilidad un valor adicional para indicar un error. Nos veríamos obligados a devolver valores especiales, como `-1` o una cadena de longitud cero. Esta metodología no es general.

## Excepciones y objetos

¿Qué tipo de elemento permite al programador indicar que ocurrió un error y permite que el error se detecte en otra región del programa? Para el principiante es tentador (pero incorrecto) suponer que esto se podría realizar mediante el uso de variables `boolean`, en la siguiente forma:

```
boolean ocurrióError = false;
...código que afecta a casoError
if (ocurrióError == true) {
 manejar el problema...
```

Sin embargo, ésta no es la forma como se indican los errores en Java. En vez de usar variables empleamos una metodología de objetos. Cuando deseamos indicar una excepción, usamos `new` para crear una instancia de una clase de excepción apropiada. Así, otra región del programa puede entonces

comprobar su existencia. Para resumir, creamos una instancia de una clase de excepción para indicar que ocurrió un error.

## Cuándo usar excepciones

Las excepciones proveen un tipo de estructura de control; entonces, ¿cuándo debemos usarlas en vez de `if` o `while`?

En definitiva, si vamos a usar clases ya existentes que se hayan escrito para producir excepciones, entonces necesitamos manejarlas. Pero consideremos esta situación: tenemos que escribir un programa para sumar una serie de números positivos que el usuario va a introducir mediante el teclado. Para indicar el final de la secuencia se introducirá `-1`. ¿Es `-1` una excepción? No; se espera como parte de la entrada normal. Una situación similar ocurre cuando nos encontramos el final del archivo al momento de leerlo. Esto no se debe considerar como una excepción. Para volver a nuestro problema de los números, la solución correcta (en español informal/Java) es de la siguiente forma:

```
obtener número;
while (número >= 0) {
 suma = suma + número;
 obtener número;
}
```

En resumen, utilizamos las herramientas de manejo de excepciones para los errores en vez de hacerlo para los casos normales.

## La jerga de las excepciones

Java tiene su propia terminología para las excepciones. Al indicar una excepción se dice que fue lanzada, y al detectarla en cualquier parte, que fue atrapada. Java cuenta con las palabras clave `throws`, `throw`, `try` y `catch` para llevar a cabo estas tareas. Primero veremos el caso más simple sobre cómo atrapar una excepción lanzada por una clase de biblioteca.

## Un ejemplo con try-catch

Ahora le presentaremos un programa simple para duplicar números, el cual invita al usuario a introducir un entero en un campo de texto y después muestra el valor duplicado en otro campo de texto, o muestra un cuadro de mensaje si se introdujo un valor no entero. Si la cadena de entrada contiene un valor que no sea dígito, el método `parseInt` no podrá procesarlo e indicará esta situación lanzando una excepción. Vamos a ver las nuevas características incorporadas en este programa en particular y después pasaremos a los casos generales.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
```

```
public class DemoExcepciones1 extends JFrame implements ActionListener {

 private JTextField campoEntrada;
 private JTextField campoResultado;
 private JLabel etiquetaResultado, etiquetaEntrada;
 private JButton botónDuplicar;

 public static void main(String[] args) {
 DemoExcepciones1 marco = new DemoExcepciones1();
 marco.setSize(300, 150);
 marco.crearGUI();
 marco.setVisible(true);
 }

 private void crearGUI() {
 setDefaultCloseOperation(EXIT_ON_CLOSE);
 Container ventana = getContentPane();
 ventana.setLayout(new FlowLayout());

 etiquetaEntrada = new JLabel("Entero: ");
 ventana.add(etiquetaEntrada);

 campoEntrada = new JTextField(10);
 ventana.add(campoEntrada);

 botónDuplicar = new JButton("Duplicar");
 ventana.add(botónDuplicar);
 botónDuplicar.addActionListener(this);

 etiquetaResultado = new JLabel("El valor duplicado es: ");
 ventana.add(etiquetaResultado);

 campoResultado = new JTextField(10);
 ventana.add(campoResultado);
 }

 public void actionPerformed(ActionEvent event) {
 if (event.getSource() == botónDuplicar) {
 campoResultado.setText("");
 try{
 int número = Integer.parseInt
 (campoEntrada.getText());
 campoResultado.setText(Integer.toString(2*número));
 }
 catch (NumberFormatException objetoError) {
 JOptionPane.showMessageDialog(null,
 "Error en número: vuelva a escribir");
 }
 }
 }
}
```



**Figura 16.1** (a) Pantalla de `DemoExcepciones1` (sin error). (b) Pantalla de `DemoExcepciones1` (con un mensaje).

Las figuras 16(a) y (b) muestran la ejecución de este programa con la entrada correcta y después con la entrada incorrecta que provoca una excepción.

La parte clave de este programa es:

```
try{
 int número = Integer.parseInt
 (campoEntrada.getText());
 campoResultado.setText(Integer.toString(2*número));
}
catch (NumberFormatException objetoError) {
 JOptionPane.showMessageDialog(null,
 "Error en número: vuelva a escribir");
}
```

Aquí es donde decimos “si ha ocurrido algo malo dentro de `parseInt`, ¡manéjalo!”

Aquí hay una nueva instrucción, la cual en esencia es una estructura de control. Es de la siguiente forma:

```
try {
 una serie de instrucciones;
}
catch (UnaExcepción objetoError) {
 maneja la excepción;
}
```

En Java, a un grupo de instrucciones encerradas entre los caracteres { y } se le conoce como “bloque”. Vamos a hablar sobre el “bloque try” y el “bloque catch”.

La idea es instruir a Java para que trate de ejecutar un bloque de instrucciones. Si se ejecuta sin producir una excepción, se ignora el bloque **catch** y la ejecución continúa después de este bloque. Pero si se produce una excepción, podemos especificar que se ejecute el bloque **catch** indicando la clase de excepción que deseamos atrapar. En nuestro ejemplo consultamos la documentación de la biblioteca para **parseInt** y descubrimos que se puede producir (o lanzar) una excepción de la clase **NumberFormatException**. Si ocurre algún otro tipo de excepción, no se ejecutará nuestro bloque “catch” y Java tratará de buscar un bloque “catch” que especifique ese tipo de excepción. A continuación describiremos este proceso con más detalle. En nuestro ejemplo tenemos lo siguiente:

```
catch (NumberFormatException objetoError) {
```

lo cual es algo así como la declaración de un método. Java deposita un objeto de tipo **NumberFormatException** en el parámetro, el cual decidimos nombrar **objetoError**. Podríamos utilizarlo si se requiriera, para lo cual usaríamos el método **toString** como en el siguiente ejemplo:

```
catch (NumberFormatException objetoError) {
 JOptionPane.showMessageDialog(null,
 "Error: " + objetoError.toString());
}
```

El método **toString** devuelve el nombre de la excepción junto con un posible mensaje adicional para aclarar su significado. Las instrucciones anteriores producen el siguiente mensaje:

```
Error: java.lang.NumberFormatException
```

Una vez que se ejecuta el bloque **catch**, la ejecución continúa debajo de éste. En muchos casos, y como los métodos se encargan de una tarea específica que no puede continuar después de que ocurre una excepción relacionada con esa tarea, es común regresar del método como en el siguiente ejemplo:

```
private void unMétodo() {
 try {
 // bloque de código
 }
 catch(Exception objetoError) {
 // maneja la excepción
 }
} // regresa
```

En nuestro programa de ejemplo esto es justo lo que necesitamos: el manejador de excepciones muestra un mensaje de error y continúa, regresando del método. El usuario puede entonces introducir un nuevo número en el cuadro de texto.

## try y los alcances

Como dijimos, cuando un bloque **try** produce una excepción, su ejecución termina (se abandona). Una consecuencia de ello es que las variables declaradas en su interior se vuelven inaccesibles y, en

especial, no se pueden utilizar en el bloque `catch`, aun cuando están en el mismo método. Esto puede ser un problema si deseamos usar esas variables para producir un mensaje de error específico. Por ejemplo, tal vez queramos mostrar un mensaje de alerta si no se pudiera convertir en un entero. Por fortuna la solución es simple: cualquier variable requerida tanto en el bloque `try` como en el bloque `catch` se debe declarar fuera del bloque `try`. Debemos hacer lo siguiente:

```
String s; // disponible para ambos bloques, try y catch
try {
 // código que usa la variable s
}
catch (Exception objetoError) {
 JOptionPane.showMessageDialog(null, "Error: s es " + s);
}
```

La alternativa es la siguiente, que no se compilará. La variable `s` es local para el bloque `try` solamente, no para el bloque `catch`.

```
try {
 String s;
 // código que usa la variable s
}
catch (Exception objetoError) {
 JOptionPane.showMessageDialog(null, "Error: s es "+ s); // No
}
```

## PRÁCTICA DE AUTOEVALUACIÓN

- 16.2** Investigue los nombres de las excepciones que podrían lanzar los métodos de la clase `Integer`.

### La búsqueda de un bloque `catch`

¿Qué pasa si el programa no atrapa una excepción lanzada? Las reglas precisas para esto dependen de la clase de excepción, como veremos más adelante. El principio básico se fundamenta en el hecho de que todos los programas anteriores tienen unas cuantas líneas compuestas de métodos. Por ende, en tiempo de ejecución se hace una llamada a un método inicial, el cual a su vez llama a otros métodos, que a su vez... El uso de métodos en tiempo de ejecución es jerárquico (un método de nivel superior llama a los métodos de nivel más bajo, etc.) y este patrón de llamadas es impredecible antes del tiempo de ejecución; esto se debe a que la decisión sobre llamar o no a un método podría depender de una instrucción `if` con base en los datos de entrada.

Imagine que `métodoA` llama a `métodoB`, el cual a su vez llama a `métodoC`. Si ocurre una excepción en `métodoC`, la búsqueda de un bloque `catch` apropiado empieza en `métodoC`. Si no se encuentra uno, la búsqueda pasa a `métodoB`. Si este último no proporciona un bloque `catch`, entonces la búsqueda pasa a `métodoA`, y si el método de nivel superior no provee un bloque `catch`, aparece un mensaje de excepción en la pantalla.

El no poder atrapar una excepción no siempre significa que el programa terminará después de mostrar el mensaje de la excepción. Esto depende de la naturaleza del programa. He aquí los casos:

- Si el programa es una aplicación de consola sin GUI (como veremos en el capítulo 17), entonces el programa sí termina.
- Si el programa tiene una GUI y la excepción ocurre al momento de crear la GUI (por ejemplo, cuando se están agregando componentes a la pantalla), el programa también termina. Una causa común de esto es omitir la creación de una instancia de un componente con `new`. Una vez que el programa de la GUI está en funcionamiento, las excepciones **no** terminan el programa. Éste continúa pero no se puede confiar en su operación. Por ejemplo, los métodos pueden pasar valores de retorno incorrectos.

## Lanzamiento de excepciones: una introducción

La mayoría de los métodos que escribirá no necesitan crear y lanzar excepciones. La cuestión principal que le debe preocupar es la de atrapar las excepciones que se lanzan desde los métodos de biblioteca. Sin embargo, para que usted adquiera un conocimiento más amplio, le mostraremos la forma en que se lanzan muchas excepciones. Recuerde la forma en que usamos `parseInt` en el primer ejemplo. No necesita examinar el código fuente de las bibliotecas (basta con su documentación), pero he aquí parte del código de la clase `Integer`:

```
public static int parseInt(String s)
 throws NumberFormatException {
 ...código para el método...
 // se detectó un error:
 throw new NumberFormatException();
 ...etc
}
```

Observe el uso de `throws` en el encabezado del método y de `throw` en el cuerpo del método. A menudo la instrucción `throw` se ejecuta como resultado de un `if`: provoca que se abandone el método actual y empieza la búsqueda de un bloque `catch` que coincida. Cuando empiece a usar los métodos provistos (busque siempre código existente antes de escribir su propio código), examinará un manual de referencia de la biblioteca de Java, ya sea en formato de libro, en su entorno de desarrollo o en un sitio Web. La información que proveerá dicha documentación es:

- Una descripción breve del propósito del método.
- Su nombre, tipos de parámetros y tipo de valor de retorno.
- Las clases de excepciones que puede lanzar.

Nuestro siguiente paso para prepararnos a usar un método es considerar con mayor detalle todas las excepciones que pueda lanzar.

## ● Clases de excepciones

En esta sección exploraremos las variedades de clases de excepciones incluidas en la biblioteca de Java. Básicamente, la biblioteca nos proporciona una lista de nombres de clases de excepciones, pero si no hay una excepción adecuada podemos crear nuestra propia excepción. No trataremos el proceso de cómo inventar nuevas excepciones pues confiamos en que usted podrá encontrar una dentro de la biblioteca con un nombre apropiado.

La herencia se utiliza para clasificar los errores en tipos distintos; por ejemplo, hay excepciones que se clasifican como “graves aprietos” (como `OutOfMemoryError`) y hay excepciones que son menos graves (como `NumberFormatException`). Hay una gran cantidad de clases de excepciones predefinidas; en la figura 16.2 aparecen las principales y se muestra su estructura de herencia, es decir, la jerarquía de clases.

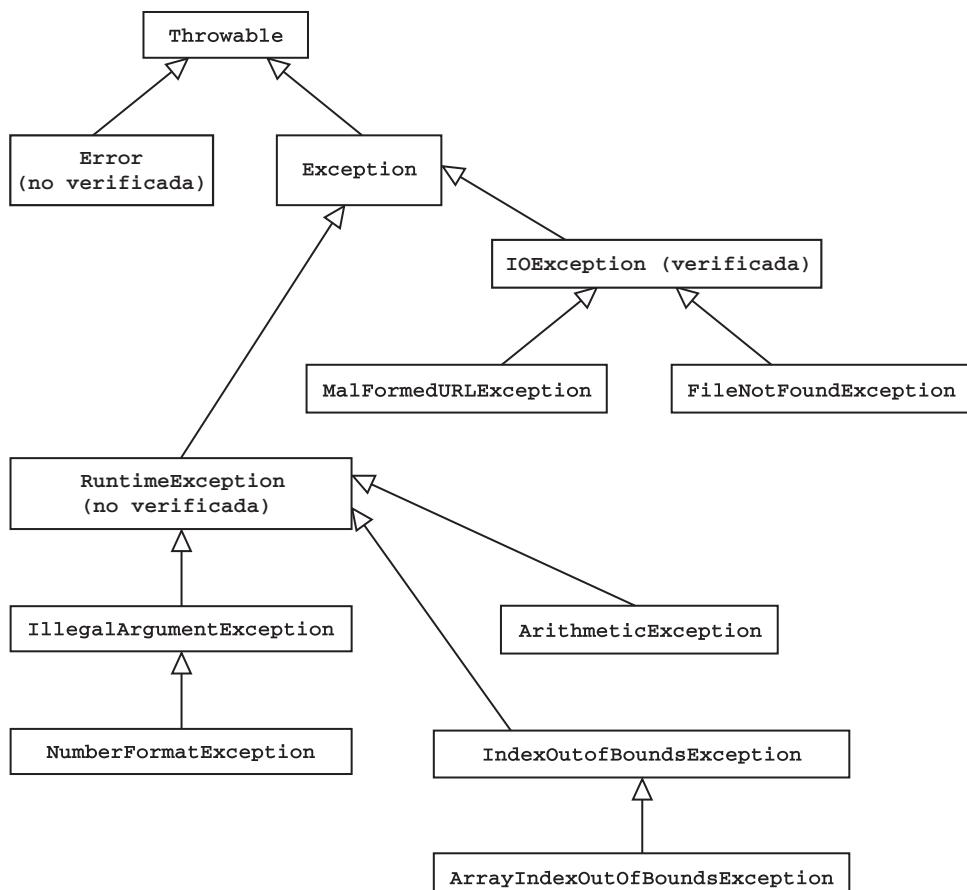
La documentación de la biblioteca nos indica el tipo de excepciones que puede lanzar un método. Para asociar una excepción con la jerarquía de la figura, tal vez tengamos que recorrer el árbol hacia arriba para buscar a la madre (es decir, la clase que extiende la excepción), y a la madre de la madre. Por ejemplo, nuestro familiar método `parseInt` de la clase `Integer` especifica que lanza una excepción `NumberFormatException`. De la figura 16.2 podemos ver que esta excepción se deriva de `IllegalArgumentException`, la cual está dentro de la clase no verificada `RuntimeException`.

Con base en la posición de una excepción en la jerarquía es posible averiguar cómo se puede procesar. En esencia, todas las excepciones están debajo de la clase `Throwable`. Hay dos subclases principales: `Error` y `Exception`. La clase `Error` tiene excepciones graves pero difíciles de corregir; no debemos preocuparnos por atraparlas. La clase `Exception` es más relevante. La mayoría de las excepciones dentro de esta clase son *verificadas*, lo cual significa que tenemos que lidiar con ellas; nuestro programa no se compilará si las ignoramos. La única que no se apega a esta regla es la clase `RuntimeException`, la cual es no verificada; su programa se compilará aún si la ignora. Ahora vamos a examinar estos conceptos con más detalle.

## ● La compilación y las excepciones verificadas

El compilador realiza comprobaciones rigurosas a la hora de relacionar las instrucciones `throw` y `try-catch`. Hay tres casos:

- Un método declara que puede lanzar una clase de excepción que debe ser aceptada por el que lo llamó, ya sea atrapándola o declarando que se puede lanzar (propagar hacia arriba). Si se omite



**Figura 16.2** Estructura de herencia de las principales excepciones..

la excepción, se producirá un error de compilación. En otras palabras, algunos métodos que utilice lo obligarán a usar `try-catch`.

- Un método declara que puede lanzar una clase de excepción que no se necesita manejar. Usted decide si usa `try-catch` o no. Si usted no maneja esa excepción, no se ignora sino que se pasa hacia arriba al método que llamó a su método (y después al método que llamó a ese método, etc.) para tratar de encontrar el primer `catch` que coincide. Se produce una coincidencia si un `catch` es específico a esa excepción en particular o a cualquier clase superior a ella. Si no existe un `catch` apropiado, el sistema de Java mostrará un mensaje de excepción.
- El caso final implica el lanzamiento de excepciones que no se indican en el encabezado de un método. He aquí el razonamiento: el uso de (por ejemplo) la aritmética de enteros y arreglos es

muy frecuente en toda la biblioteca de Java y en el código que usted escribe. Dicho código puede producir errores (por ejemplo, `ArithmeticException` en el caso de una división entera entre cero y `ArrayIndexOutOfBoundsException` en la clase `IndexOutOfBoundsException` cuando el valor del índice de un arreglo está fuera del rango declarado). Ambas excepciones están en la clase `RuntimeException`. Como muchos métodos pueden producir estas situaciones, no se declaran y el programador es quien decide si utiliza o no `try-catch`.

## Cómo atrapar excepciones – los casos comunes

Analizaremos las posibles alternativas con tres casos comunes:

- La excepción verificada `FileNotFoundException`.
- La excepción no verificada `NumberFormatException`.
- La excepción no verificada `ArrayIndexOutOfBoundsException`.

Una excepción `FileNotFoundException` (de la clase `IOException`) se produce cuando un programa trata de acceder a un archivo que no existe; tal vez nos equivocamos en el nombre o está en la carpeta equivocada, o tal vez insertamos el dispositivo de memoria equivocado. En el capítulo 17 veremos los detalles del acceso a los archivos, pero todo lo que tenemos que saber en este momento es que, antes de leer datos de un archivo, lo “abrimos” mediante una instrucción de la forma:

```
archivoEnt = new BufferedReader(new FileReader("c:\\miarchivo.txt"));
```

en donde `miarchivo.txt` es el nombre del archivo que el programa espera que esté presente. Sin embargo, el constructor de `FileReader` lanza una excepción verificada con la que debemos lidiar: podemos usar `try-catch` o especificar que el método que contiene la instrucción anterior puede lanzar la excepción.

En los programas en los que un usuario introduce el nombre del archivo, la primera alternativa es mejor: lidiar inmediatamente con la excepción, pidiéndole al usuario que seleccione otra cosa, en vez de complicarle la existencia a otros métodos. La estrategia es básicamente igual a la de nuestro ejemplo para duplicar números en el que utilizamos la excepción `NumberFormatException`. Usamos lo siguiente:

```
try {
 archivoEnt = new BufferedReader(new
 FileReader("c:\\miarchivo.txt"));
}
catch (FileNotFoundException objetoError) {
 JOptionPane.showMessageDialog(null,
 "Error: no se encontró el archivo - intente de nuevo");
}
```

La segunda alternativa es pasar la excepción al método que hizo la llamada. Creamos el siguiente método:

```
private void abrirArchivo() throws IOException {
 archivoEnt = new BufferedReader(new
 FileReader("c:\\miarchivo.txt"));
}
```

Con esta alternativa, el método que llama a `abrirArchivo` debe atrapar la excepción o pasársela hacia arriba.

Nuestro segundo ejemplo, en donde `parseInt` lanza una excepción `NumberFormatException`, es básicamente similar: atrapamos la excepción o la podemos pasársela hacia arriba. La diferencia del caso anterior de `FileNotFoundException` es que el compilador no nos obliga a manejar la excepción, ya que `NumberFormatException` (de la clase `RuntimeException`) es no verificada.

¿Qué opciones tenemos? La opción más simple es ignorar la excepción de manera intencional. El compilador no nos obligará a verificarla. Sin embargo, la única ocasión en que sería sensato hacer esto es cuando sabemos que la cadena a convertir en un entero está libre de errores (tal vez se haya recibido como entrada de un archivo correcto o se haya generado dentro del programa). Pero para el software interactivo basado en GUI, es esencial que atrapemos los errores potenciales de entrada como hicimos en el programa para duplicar números.

Al igual que con nuestra excepción de archivo, podríamos especificar que el método que llama a `parseInt` puede lanzar la excepción `NumberFormatException`, para así obligar a los métodos que hacen la llamada a atrapar la excepción o pasársela de manera explícita hacia arriba. Esta estrategia no tiene ninguna ventaja, por lo que no la analizaremos.

Nuestro ejemplo final con `ArrayIndexOutOfBoundsException` no es tan simple. Está bien atrapar la excepción, pero ¿después qué? Los ejemplos anteriores involucran la entrada por parte del usuario y se podrían corregir invitando al usuario a que lo intente otra vez. He aquí un fragmento de programa con una excepción:

```
int a[] = new int[10]; // 0 a 9, inclusive
for (int n = 0; n <= 10; n++) {
 a[n] = 0;
}
```

La instrucción `n <= 10` produce un ciclo de 0 a 10 inclusive, y por ende intenta acceder a `a[10]`; deberíamos haber puesto `n<10`. Al igual que con todas las excepciones de la clase `RuntimeException`, involucra un error de programación y para corregirlo tenemos que depurar y volver a compilar.

Y, ¿qué hacemos con una excepción `RuntimeException`? En la gran mayoría de los casos necesitamos que se indique el origen de la excepción, para que se pueda rastrear y corregir. Podemos hacer esto fácilmente si ignoramos las excepciones, lo cual el compilador nos permite hacer con la clase `RuntimeException`, ya que es no verificada. La excepción se pasa hacia arriba y se muestra un mensaje que indica el nombre de la excepción seguido de un rastreo de las llamadas a los métodos, con los números de línea relacionados con el código fuente de Java.

Ya hemos cubierto las excepciones con el detalle suficiente para el principiante, por lo que sería sensato que se detuviera aquí. Lo que veremos a continuación es para el programador más experimentado.

## Uso de la estructura de las clases de excepciones

La figura 16.2 muestra la estructura de herencia de las excepciones. Al referirnos a una clase madre, podemos atrapar cualquier excepción de esa clase. Por ejemplo, podríamos usar:

```
try {
 // algo de código
}
catch (Exception objetoError) {
 // manejarla...
}
```

con lo cual atraparíamos excepciones de `RuntimeException`, `IOException`, etc., debido a que `Exception` es su superclase. Si deseamos tratar a las excepciones individuales de manera especial, pero queremos atrapar a todas las de una clase específica, podemos usar lo siguiente:

```
try {
 // algo de código
}
catch (IOException objetoError) {
 // manejar problema de E/S
}
catch (Exception objetoError) {
 // manejar las demás excepciones
}
```

El compilador de Java requiere que pongamos la clase más general al final. Si invertimos el orden de las dos instrucciones `catch` anteriores se producirá un error de compilación.

## Fundamentos de programación

- Al escribir métodos de propósito general (en donde tal vez no conozcamos el uso que se les vaya a dar en un futuro) debemos lanzar excepciones en vez de terminar el programa, ocultar un posible error o producir resultados incorrectos.
- Una excepción cambia el orden en el que se obedecen las instrucciones; por lo tanto se considera como una forma de estructura de control. Pero las excepciones no se deben utilizar para casos normales en los que las estructuras de control convencionales puedan lidiar adecuadamente con el problema.

## Errores comunes de programación

- Si no aceptamos una excepción verificada se produce un error de compilación. Debemos atrapar la excepción en forma local o especificar que nuestro método la va a lanzar.
- Permitir el lanzamiento de una excepción desde un método cuando es evidente que se puede manejar en forma local.
- Declarar un elemento en un bloque `try` e intentar hacer referencia a ese elemento en un bloque `catch`. Dichos elementos se deben declarar arriba (fuera) del bloque `try`.
- Tratar de atrapar y procesar la clase `RuntimeException`. En la mayoría de los casos no se puede hacer mucho. Al atrapar esos errores se podrían ocultar, cuando en realidad el programador necesita saber que están presentes.

## Secretos de codificación

La forma básica del bloque `try-catch` es:

```
try {
 una serie de instrucciones;
}
catch (UnaExcepción objetoError) {
 manejar la excepción;
}
```

## Nuevos elementos del lenguaje

- `try`.
- `catch`.
- La jerarquía de clases de excepciones.

## Resumen

- Una excepción es una situación inusual.
- Las excepciones son instancias de clases, creadas con `new`.
- Cuando se detecta una situación excepcional, se lanza una excepción.
- Se utilizan bloques `try-catch`. Rodeamos el código que podría lanzar una excepción mediante `try { }` y proveemos un bloque `catch` que coincida.
- Una excepción verificada debe aceptarse.
- El árbol de herencia de la clase `Throwable` (figura 16.2) es imprescindible, ya que nos muestra las principales excepciones con las que tendremos que lidiar y en qué clase se encuentran.
- Las excepciones dentro de la clase `RuntimeException` son difíciles de corregir; en muchos casos se pueden ignorar en forma intencional. Las demás excepciones debajo de la clase `Exception` son verificadas, por lo cual tenemos que lidiar con ellas.

## Ejercicios

- 16.1** Escriba un programa que proporcione dos campos de texto para introducir los valores enteros `a` y `b`. Muestre el resultado de `a / b`. Incorpore el manejo de excepciones para los campos de texto: muestre un mensaje si se utiliza entrada que no conste de números enteros.
- 16.2** Modifique el ejercicio 16.1 de manera que maneje valores `double`.
- 16.3** Escriba un programa que calcule el interés de un año sobre un monto inicial. El usuario introduce el monto inicial y el interés por año como un valor porcentual (valores `double`). Debe proporcionar el manejo de excepciones para una entrada que no sea numérica.
- 16.4** Consulte el ejercicio 16.3. Realice la misma tarea, pero escriba un método:

```
private double interés(double inicial,
 double porAño)
```

que devuelva el interés. Agregue el manejo de excepciones al método para verificar que todos los valores sean positivos. El encabezado del método debe usar `throws` con `NumberFormatException` y el cuerpo debe incorporar una instrucción `throw`.

- 16.5** Si conocemos la longitud de los tres lados de un triángulo, podemos calcular el área mediante la siguiente fórmula:

```
área = Math.sqrt(s * (s - a) * (s - b) * (s - c));
```

en donde:

```
s = (a + b + c) / 2;
```

Escriba un método para calcular y devolver el área. Haga que lance una excepción cuando las tres longitudes no puedan formar un triángulo.

Escriba un método que llame a su método para calcular el área, el cual debe atrapar su excepción.

- 16.6** Escriba un programa que reciba como entrada una cadena de un campo de texto, la cual representa una fecha en la forma MM/DD/AA (por ejemplo, 03/02/01). Use `StringTokenizer` para dividirla y produzca un mensaje de error si un elemento no es numérico, si no se introdujo o si especifica una fecha imposible. Ignore los años bisiestos.

## Respuestas a las prácticas de autoevaluación

- 16.1** En los primeros dos casos, salir del programa no sería un buen curso de acción. Una respuesta más útil sería mostrar algún tipo de indicación de error y permitir al usuario intentarlo otra vez o abandonar la selección del elemento.

En el tercer caso, la complicación es que la impresora se podría quedar sin papel a mitad de una impresión. Es necesario informar al usuario sobre esto y tal vez proporcionarle opciones para abandonar la solicitud de impresión o, suponiendo que se haya cargado más papel, seguir imprimiendo desde una página específica.

- 16.2** Una búsqueda en la documentación revela que `NumberFormatException` es la única excepción.

# CAPÍTULO **17**



## Archivos y aplicaciones de consola

En este capítulo conoceremos:

- Qué es un archivo de texto.
- Cómo leer y escribir datos en archivos.
- Cómo manipular rutas y nombres de carpetas.
- Cómo escribir aplicaciones de consola.

### ● Introducción

Ya hemos trabajado con archivos en capítulos anteriores, puesto que utilizamos un editor para crear archivos de código fuente de Java y un sistema operativo para ver una estructura jerárquica de directorios (carpetas). En este capítulo analizaremos la naturaleza de la información que podemos guardar en los archivos y cómo podemos escribir programas para manipularlos. Primero vamos a aclarar la diferencia entre la RAM y los dispositivos de almacenamiento de archivos (por ejemplo, discos, unidades de disco duro y CD-ROM). La capacidad de almacenamiento se mide en bytes en todos estos casos. Hay varias diferencias importantes entre la RAM y el almacenamiento de archivos:

- El tiempo para acceder a un elemento en la RAM es mucho menor. El costo de la RAM es mayor (megabyte por megabyte).
- Los datos en la RAM son temporales: se borran al apagar el equipo. Los dispositivos de almacenamiento de archivos pueden guardar los datos en forma permanente.
- La RAM se utiliza para almacenar los programas a medida que se ejecutan. Se copian de un dispositivo de almacenamiento de archivos justo antes de su ejecución.

La capacidad de los dispositivos de almacenamiento de archivos es mayor. Los CD-ROM tienen una capacidad de alrededor de 650 megabytes y los DVD (Discos Versátiles Digitales) tienen una capacidad de 4.7 gigabytes (1 gigabyte = 1024 megabytes). Tanto los CD-ROM como los DVD tienen versiones regrabables.

Los discos duros comunes pueden guardar alrededor de 700 gigabytes. Sin embargo, la tecnología evoluciona con rapidez; el objetivo es crear dispositivos de almacenamiento económicos, rápidos y pequeños que el software de computadora moderno requiere, en especial en el área de las imágenes fijas, las imágenes en movimiento y el sonido de alta calidad.

## ● Acceso a los archivos: ¿flujo o aleatorio?

Java cuenta con más de 20 clases para acceder a los archivos, cada una de las cuales tiene su propio conjunto de métodos. Pero con una opción tan amplia, ¿qué clases deberíamos elegir? Una decisión importante implica la elección entre el acceso por flujo y el acceso aleatorio. Cuando usamos el acceso por flujo en un archivo, debemos tratarlo como una secuencia de elementos que se deben procesar uno después del otro, empezando con el primero. Para muchas tareas esto es completamente apropiado. Si usamos el acceso aleatorio, podemos saltar de inmediato a una posición de byte específica en un archivo. En ciertas aplicaciones (como en las bases de datos) esto puede agilizar el procesamiento, pero también es más complicado de programar. En realidad es más probable que utilice una biblioteca de clases de bases de datos en vez de codificar su propio acceso al disco de bajo nivel. Por esta razón nos enfocaremos en los flujos.

## ● Aspectos esenciales de los flujos

Primero le presentaremos la jerga, que es similar en la mayoría de los lenguajes de programación. Si deseamos procesar los datos en un archivo existente, debemos:

1. Abrir el archivo.
2. Leer o recibir como entrada los datos, elemento por elemento, y colocarlos en variables.
3. Cerrar el archivo cuando terminemos de trabajar con él.

Para transferir datos de variables a un archivo, debemos:

1. Abrir el archivo.
2. Enviar como salida (o escribir) nuestros elementos en la secuencia requerida.
3. Cerrar el archivo cuando terminemos de trabajar con él.

Cabe mencionar que, al leer datos de un archivo, todo lo que podemos hacer es leer el siguiente elemento que se encuentre en él. Si, por ejemplo, necesitamos examinar el último elemento en un archivo, tendremos que codificar un ciclo para leer un elemento a la vez hasta llegar al elemento requerido. Para muchas tareas es conveniente visualizar un archivo de texto como una serie de líneas de texto, cada una compuesta de varios caracteres. Cada línea se termina mediante un carácter de fin de línea. En este capítulo utilizaremos las clases de Java que nos permiten acceder a un archivo

línea por línea. Uno de los beneficios de esta metodología es que resulta sencillo transferir archivos entre aplicaciones. Por ejemplo, podría crear un archivo al ejecutar un programa de Java y después cargarlo en un procesador de palabras, editor de texto o paquete de correo electrónico.

## ● Las clases de E/S de Java

Las clases de flujos están organizadas en forma jerárquica. He aquí las clases de entrada más útiles:



Las clases **Reader** y **Writer** están en la parte superior de un árbol de herencia y se extienden a través de varias clases basadas en caracteres. Para los archivos, vamos a usar las clases **BufferedReader** y **PrintWriter** para leer y escribir líneas de texto, y para una entrada que no sea de archivo (del teclado y de páginas Web) vamos a usar también **InputStreamReader**. Como verá en nuestros ejemplos, los programas que utilizan archivos deben importar **java.io.\***.

A propósito, el uso del término *búfer* (*buffer*) significa que tras bambalinas el software lee un gran trozo de datos del lento dispositivo de almacenamiento de archivos (CD-ROM o disco duro) y lo almacena en la RAM de alta velocidad. Las llamadas sucesivas de los métodos que necesitan leer una pequeña cantidad de datos del dispositivo de almacenamiento de archivos pueden obtener con prontitud los datos de la RAM. Por ende, un búfer actúa como un amortiguador entre el dispositivo de almacenamiento y el programa.

## ● Las clases **BufferedReader** y **PrintWriter**

Para leer y escribir líneas de texto, utilizaremos:

- El método **readLine** de **BufferedReader**. Este método lee toda una línea de texto de una cadena. Si necesitamos dividir la línea en partes separadas, podemos usar la clase  **StringTokenizer** que vimos en el capítulo 15.
- La clase **PrintWriter**. Esta clase tiene dos métodos principales: **print** y **println**. Observe la **l** minúscula. Ambos métodos escriben una cadena en un archivo, pero **println** agrega el carácter de fin de línea después de la cadena. Si necesitamos crear una línea a partir de partes separadas, podemos usar el operador de cadenas **+** para unir subcadenas.

## ● Salida de archivos

En esta sección vamos a presentar un programa que nos permite escribir el nombre de un archivo, algo de texto y después guardar el texto en un archivo. Los elementos básicos de la interfaz de usuario son:

- Un campo de texto para aceptar el nombre del archivo.
- Un área de texto desplazable para aceptar el texto del usuario (recuerde que un área de texto nos permite cortar, copiar y pegar).
- Un botón “guardar” para iniciar la transferencia de texto del área de texto al archivo.

He aquí la aplicación completa. La figura 17.1 la muestra en ejecución.

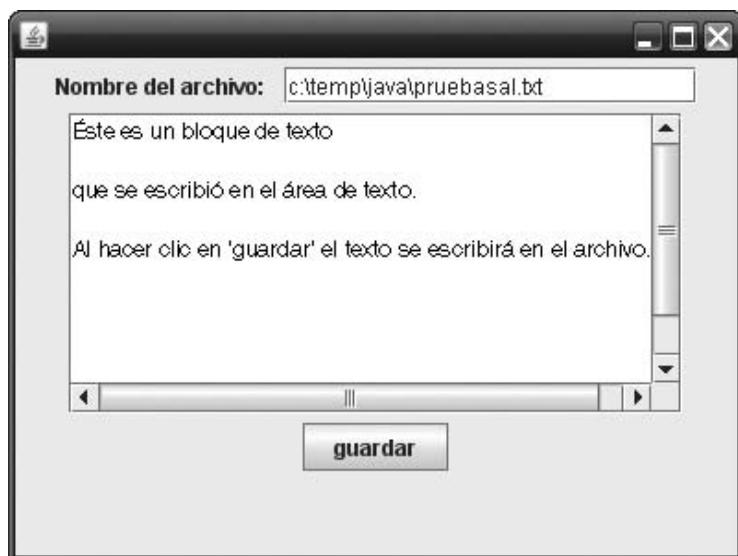


Figura 17.1 Pantalla del programa `DemoSalidaArchivos`.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.util.*;
import java.io.*;

public class DemoSalidaArchivos extends JFrame
 implements ActionListener {

 private JTextArea areaTexto;
 private JButton botonGuardar;
 private JTextField campoNombre;
 private JLabel etiquetaNombre;
 private PrintWriter archivoSalida;
```

```

public static void main(String [] args) {
 DemoSalidaArchivos marco = new DemoSalidaArchivos();
 marco.setSize(400, 300);
 marco.createGUI();
 marco.setVisible(true);
}

private void createGUI() {
 setDefaultCloseOperation(EXIT_ON_CLOSE);
 Container ventana = getContentPane();
 ventana.setLayout(new FlowLayout());

 etiquetaNombre = new JLabel("Nombre del archivo: ");
 ventana.add(etiquetaNombre);

 campoNombre = new JTextField(20);
 ventana.add(campoNombre);

 areaTexto = new JTextArea(10,30);
 JScrollPane panelDespl = new JScrollPane(areaTexto);
 ventana.add(panelDespl);

 botonGuardar = new JButton("guardar");
 ventana.add(botonGuardar);
 botonGuardar.addActionListener(this);
}

public void actionPerformed(ActionEvent event) {
 if (event.getSource() == botonGuardar) {
 try{
 archivoSalida = new PrintWriter(
 new FileWriter(campoNombre.getText()), true);
 archivoSalida.print(areaTexto.getText());
 archivoSalida.close();
 }
 catch (IOException e) {
 JOptionPane.showMessageDialog(null,
 "Error de archivo: " + e.toString());
 }
 }
}
}

```



Primero declaramos lo siguiente:

```
private PrintWriter archivoSalida;
```

Podríamos haber hecho esa variable local para `actionPerformed`, pero en programas más grandes tal vez varios métodos necesiten acceder al archivo. Después creamos una instancia:

```
archivoSalida = new PrintWriter(
 new FileWriter(campoNombre.getText()), true);
```

Básicamente, la instrucción anterior abre el archivo cuyo nombre se escribió en el campo de texto. Como podemos ver en la figura 17.1, elegimos el archivo llamado **pruebasal.txt** en el directorio **temp** de la unidad **c:**. Para predeterminar el archivo en el programa podríamos usar lo siguiente:

```
new FileWriter("c:\\temp\\demo.txt", true);
```

El carácter **\** se utiliza para separar directorios y nombres de archivos; utilizamos un **\** adicional para escaparlo. Si el archivo no existe, se crea en ese instante; si existe, se sobreescibirá y se reemplazará con su nuevo contenido. El detalle es que primero debemos construir una instancia de **FileWriter**, cuyo constructor nos permite especificar un nombre de archivo como una cadena de texto (la clase **PrintWriter** no tiene un constructor así). Después creamos una instancia de **PrintWriter**, cuyo constructor puede aceptar dos parámetros:

- Una instancia de **FileWriter**.
- Un valor **boolean**. Si proveemos un valor **true**, el búfer interno se escribirá (“vaciará”) de manera automática al archivo cada vez que se llame a **println**. Hemos elegido esta opción denominada “autovaciado” para que el contenido del archivo se mantenga relativamente actualizado. Esto puede ayudar durante la depuración (si no especificáramos esta opción y el programa fallara, tal vez los caracteres en el búfer nunca llegarían al archivo).

Después se asigna la nueva instancia de **PrintWriter** a **archivosalida**. Hay dos elementos que al principio podríamos llegar a confundir: uno es el nombre del archivo que utiliza el sistema operativo; el otro es el nombre de una variable de flujo dentro del programa: usted tiene la libertad de elegir el nombre para este flujo.

En realidad, para escribir al flujo usamos **print**, como en el siguiente ejemplo:

```
archivoSalida.print(áreaTexto.getText());
```

Recuerde que **getText** obtiene la cadena completa del área de texto (que puede consistir en varias líneas). Después imprimimos al archivo. Por último lo cerramos:

```
archivoSalida.close();
```

Al crear el flujo de salida con **new** se podría producir una excepción verificada, por lo que tenemos que rodear el código con un bloque **try-catch** para detectar una excepción **IOException** (o una excepción más específica que la extienda). Además, muchas de las clases de E/S pueden lanzar excepciones al leer o escribir, por lo que hemos extendido el rango del bloque **try** para cubrir las llamadas de **print** y **close**.

Por desgracia somos culpables de ignorar un caso especial aquí: **PrintWriter** es muy inusual debido a que los potenciales errores de **print** sólo se pueden detectar mediante la llamada a un método conocido como **checkError** en vez de atrapar una excepción. En realidad deberíamos usar:

```
archivoSalida.print(áreaTextoEntrada.getText());
if (archivoSalida.checkError()) // true si hay error
 JOptionPane.showMessageDialog(null,
 "Error al escribir en el archivo.");
```

Pero como pueden ocurrir pocas excepciones al escribir cadenas en un archivo (una posibilidad sería llenar el disco), ignoramos de manera intencional el caso especial de **PrintWriter/checkError** y hacemos énfasis en un patrón para la mayoría de las clases de E/S.

En resumen, cuando el usuario hace clic en el botón “guardar”, el programa:

1. Abre el archivo seleccionado.
2. Obtiene una cadena del área de texto, la cual puede contener varias líneas.
3. Envía como salida (escribe) la cadena al archivo.
4. Cierra el archivo.

Para verificar que se haya transferido el texto, use un editor de texto para ver el archivo.

## PRÁCTICA DE AUTOEVALUACIÓN

### 17.1 Corrija el error en este código:

```
private PrintWriter archivoSalida;
archivoSalida = new PrintWriter(campoNombre.getText());
```

## Entrada de archivos

En esta sección le presentaremos un programa que muestra el contenido de un archivo en la pantalla. La interfaz de usuario tiene:

- Un área de texto desplazable para mostrar el texto que se recibirá como entrada del archivo.
- Un campo de texto para aceptar el nombre del archivo. Más adelante utilizaremos un selector de archivos, el cual permite a un usuario seleccionar un archivo mediante el proceso de explorar y hacer clic.
- Un botón “abrir” para iniciar la transferencia de texto del archivo al área de texto.

He aquí la aplicación completa. La figura 17.2 muestra una ejecución de ejemplo. Escribimos el mismo nombre de archivo que en el programa **DemoSalidaArchivos**, por lo que el texto es el mismo.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.util.*;
import java.io.*;

public class DemoEntradaArchivos extends JFrame
 implements ActionListener {

 private JTextArea areaTexto;
 private JButton botonAbrir;
 private BufferedReader archivoEntrada;
 private JTextField campoNombre;
 private JLabel etiquetaNombre;

 public static void main (String [] args) {
 DemoEntradaArchivos marco = new DemoEntradaArchivos();
```

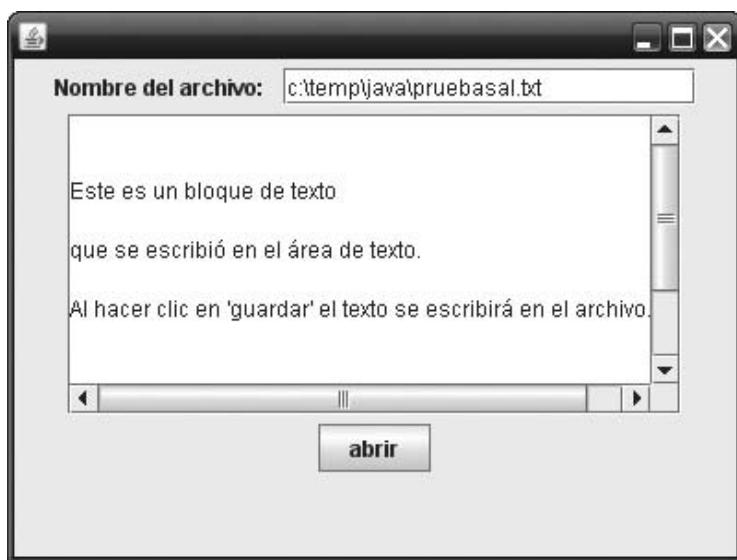


Figura 17.2 Pantalla del programa DemoEntradaArchivos.

```
marco.setSize(400, 300);
marco.crearGUI();
marco.setVisible(true);
}

private void crearGUI() {
 setDefaultCloseOperation(EXIT_ON_CLOSE);
 Container ventana = getContentPane();
 ventana.setLayout(new FlowLayout());

 etiquetaNombre = new JLabel("Nombre del archivo: ");
 ventana.add(etiquetaNombre);

 campoNombre = new JTextField(20);
 ventana.add(campoNombre);
 campoNombre.addActionListener(this);

 areaTexto = new JTextArea("",10,30);
 JScrollPane panelDespl = new JScrollPane(areaTexto);
 ventana.add(panelDespl);

 botonAbrir = new JButton("abrir");
 ventana.add(botonAbrir);
 botonAbrir.addActionListener(this);
}

public void actionPerformed(ActionEvent event) {
 if (event.getSource() == botonAbrir) {
 try {
```

```

 archivoEntrada = new BufferedReader(
 new FileReader(campoNombre.getText()));
 áreaTexto.setText(""); // borra el área de entrada
 String línea;
 while ((línea = archivoEntrada.readLine()) != null) {
 áreaTexto.append(línea+"\n");
 }
 archivoEntrada.close();
 }
 catch (IOException e) {
 JOptionPane.showMessageDialog(null,
 "Error de archivo: " + e.toString());
 }
}
}

```



Nos enfocaremos en la forma en que se usan los archivos. Primero declaramos lo siguiente:

```
private BufferedReader archivoEntrada;
```

A continuación creamos una instancia:

```
archivoEntrada = new BufferedReader(
 new FileReader(campoNombre.getText()));
```

Después utilizamos `readLine` para introducir la serie de líneas en el archivo y adjuntamos cada una de ellas a nuestra área de texto. Hay un punto crucial aquí: no sabemos cuántas líneas tiene el archivo, por lo que establecemos un ciclo que termina cuando no haya nada más que leer:

```

while ((línea = archivoEntrada.readLine()) != null) {
 áreaTexto.append(línea+"\n");
}
archivoEntrada.close();

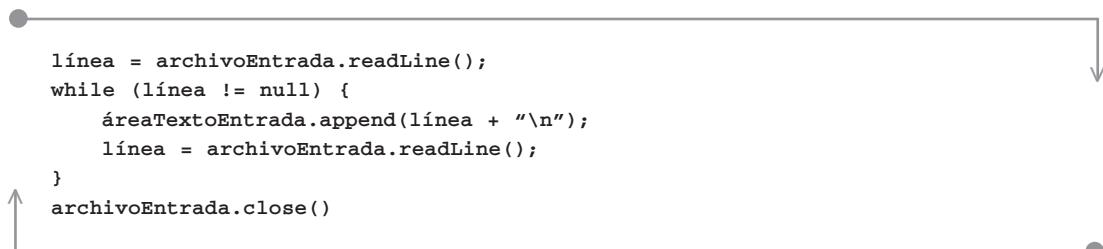
```

La condición de la instrucción `while` es bastante inusual, pero se utiliza comúnmente. Cuando `readLine` no puede encontrar más datos devuelve `null` y este valor se asigna a `línea`. Sin embargo, en Java es posible utilizar el valor asignado a una variable directamente, lo cual hacemos en este caso. Los paréntesis son esenciales para asegurar que la asignación se realice antes de la comparación. Podríamos haber puesto lo siguiente:

```

línea = archivoEntrada.readLine();
while (línea != null) {
 áreaTextoEntrada.append(línea + "\n");
 línea = archivoEntrada.readLine();
}
archivoEntrada.close()

```



Pero la versión condensada (con sus raíces en el lenguaje C) es el estilo que verá en la mayoría de los programas publicados.

En resumen, cuando el usuario hace clic en el botón “abrir”, el programa:

1. Recibe del campo de texto el nombre de un archivo como entrada.
2. Abre un archivo con este nombre.
3. Recibe como entrada las líneas del archivo y las adjunta al área de texto mientras que no llegue al fin del archivo.
4. Cierra el archivo.

## PRÁCTICAS DE AUTOEVALUACIÓN

- 17.2** Explique lo que ocurriría en el programa `DemoEntradaArchivos` si reemplazamos la línea:

```
áreaTexto.append(línea + "\n");
```

por:

```
áreaTexto.append(línea);
```

- 17.3** Explique el problema con este código, cuyo propósito es leer todas las líneas de un archivo. Suponga que se han declarado todas las variables:

```
línea = archivoEntrada.readLine();
while (línea != null) {
 áreaTexto.append(línea + "\n");
}
```

## ● Operaciones de búsqueda con archivos

Buscar un elemento en un archivo que cumpla con ciertos criterios específicos es una tarea clásica. En esta sección construiremos un programa que realiza búsquedas en un archivo de calificaciones de exámenes, que tiene la siguiente forma:

```
J.Doe, 43, 67
D.Bell, 87, 99
K.Bush, 54, 32
etc...
```

Para crear este archivo podemos escribir y ejecutar un programa de Java, o utilizar un editor de texto. Cada línea está dividida en tres áreas separadas por comas. Sin embargo, puede haber espacios adicionales. En términos de bases de datos, a dichas áreas se les conoce como campos. El programa nos permitirá escribir el nombre de un archivo y el nombre de un estudiante, el cual vamos a suponer que es único. Si los nombres no son únicos tendríamos que introducir un campo adicional para guardar un número de identificación único para cada persona. El programa buscará en el archivo y mostrará las calificaciones del estudiante que elijamos. El código que necesitamos agregar

a nuestro ejemplo anterior sobre entrada de archivos es una instrucción `while` que termine al encontrar el fin de archivo o el nombre requerido. Utilizaremos un objeto `StringTokenizer` para permitir el acceso a cada campo en turno de la línea completa.

Como hay dos formas en que puede terminar el ciclo, introduciremos una variable adicional llamada `encontró` para indicar si se encontró o no el elemento. La estructura de la búsqueda utilizando una mezcla informal de español y Java sería:

```
boolean encontró = false
while ((más líneas)) && (no encontró) {
 obtener primer campo;
 if (primer campo coincide con nombre) {
 encontró = true;
 poner el resto de los campos en campos de texto;
 }
}
```

Agregamos al programa un botón “buscar”, el cual hace que se abra el archivo y se realice la búsqueda. El usuario puede seleccionar cualquier archivo para la búsqueda. Hicimos el manejo de excepciones más específico: el programa indica cuando el nombre de un archivo no existe y cuando ocurre un error durante la lectura. He aquí el programa, que produce la pantalla de la figura 17.3:

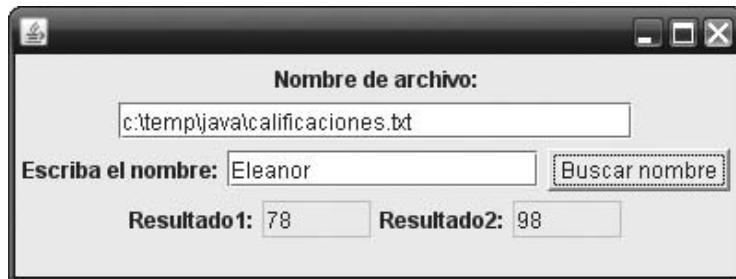


Figura 17.3 Pantalla del programa BúsquedaArchivos.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.util.*; // StringTokenizer
import java.io.*;

public class BúsquedaArchivos extends JFrame
 implements ActionListener {

 private BufferedReader archivoEntrada;
 private Button botónBuscar;
 private JTextField campoResultado1;
```

```
private JTextField campoResultado2;
private JTextField campoPersona;
private JTextField campoNombreArchivo;
private String nombreArchivo;
private JLabel etiquetaResultado1, etiquetaResultado2;
private JLabel etiquetaNombrePersona;
private JLabel etiquetaArchivo;

public static void main (String [] args) {
 BúsquedaArchivos marco = new BúsquedaArchivos();
 marco.setSize(400, 150);
 marco.createGUI();
 marco.setVisible(true);
}

private void createGUI() {
 setDefaultCloseOperation(EXIT_ON_CLOSE);
 Container ventana = getContentPane();
 ventana.setLayout(new FlowLayout());

 etiquetaArchivo = new JLabel("Nombre de archivo:");
 ventana.add(etiquetaArchivo);

 campoNombreArchivo = new JTextField(25);
 campoNombreArchivo.setText("");
 ventana.add(campoNombreArchivo);

 etiquetaNombrePersona = new JLabel("Escriba el nombre:");
 ventana.add(etiquetaNombrePersona);

 campoPersona = new JTextField(15);
 campoPersona.setText("");
 ventana.add(campoPersona);

 botónBuscar = new JButton("Buscar nombre");
 ventana.add(botónBuscar);
 botónBuscar.addActionListener(this);

 etiquetaResultado1 = new JLabel("Resultado1:");
 ventana.add(etiquetaResultado1);

 campoResultado1 = new JTextField(5);
 campoResultado1.setEditable(false);
 ventana.add(campoResultado1);

 etiquetaResultado2 = new JLabel("Resultado2:");
 ventana.add(etiquetaResultado2);

 campoResultado2= new JTextField(5);
 campoResultado2.setEditable(false);
 ventana.add(campoResultado2);
}

public void actionPerformed(ActionEvent evt) {
 if (evt.getSource() == botónBuscar) {
```

```

 campoResultado1.setText("");
 campoResultado2.setText("");
 nombreArchivo = campoNombreArchivo.getText();
 try {
 archivoEntrada = new BufferedReader(
 new FileReader(nombreArchivo));
 }
 catch (IOException e) {
 JOptionPane.showMessageDialog(null,
 "No se encontró el archivo: " + campoNombreArchivo.getText());
 return;
 }

 // ahora lee el archivo
 try {
 String linea;
 boolean encontró = false;
 while (((linea = archivoEntrada.readLine()) != null)
 && (! encontró)) {
 // los tokens se dividen en comas, espacios
 StringTokenizer tokens = new
 StringTokenizer(linea, ",");
 String nombreEnArchivo = tokens.nextToken();
 if (campoPersona.getText().equals(nombreEnArchivo)) {
 encontró = true;
 campoResultado1.setText(tokens.nextToken());
 campoResultado2.setText(tokens.nextToken());
 }
 }
 archivoEntrada.close();
 }
 catch (IOException e) {
 JOptionPane.showMessageDialog(null,
 "Error al leer el archivo "+ nombreArchivo +
 ": " + e.toString());
 }
 }
}
}

```



### PRÁCTICA DE AUTOEVALUACIÓN

- 17.4** Modifique el programa `BúsquedaArchivos` para que permita al usuario escribir un nombre en mayúsculas o minúsculas (es decir, que `john` coincida con `John` o `JOHN`). Use `equalsIgnoreCase` de la clase `String`.

## ● La clase **File**

Esta clase provee herramientas para manipular rutas de archivos y directorios (carpetas) en general. No está relacionada con el acceso a los datos dentro de los archivos. Puede utilizar la clase **File** sin necesidad de usar E/S de flujos y viceversa. Pero en cada caso necesita importar **java.io.\***.

Antes de continuar vamos a desviarnos un poco para hablar sobre las estructuras de directorio. Como sabe, los sistemas operativos proporcionan una estructura jerárquica, con una ruta a través de una estructura de la siguiente forma:

```
c:\temp\java\demo.txt
```

Ésta es una ruta estilo Windows, en la unidad **c:**, donde se utiliza el carácter \ como separador. En un sistema Unix o GNU/Linux, el separador es /.

La ruta anterior es absoluta: empieza desde la parte superior de la estructura de directorios y nos lleva hacia el archivo.

Ahora vamos a examinar un programa (llamado **DemoClaseFile**) que utiliza varios métodos de la clase **File**. Vamos a utilizar la estructura de directorios antes mencionada. Para obtener los mismos resultados que el código que mostramos, use su administrador de archivos (por ejemplo, el Explorador de Windows en un sistema Windows) y cree un nuevo directorio llamado **java** dentro del directorio **c:\temp**. Después utilice un editor de texto y teclee unas cuantas líneas de texto. Guarde el archivo como **demo.txt** en el directorio **java**.

El programa **DemoClaseFile** sólo necesita un botón “iniciar”, por lo que no mostraremos una pantalla de ejemplo. He aquí el código para manejar eventos:

```
if (event.getSource() == botónIniciar) {
 File miArchivo = new File("c:\\temp\\java\\demo.txt");
 String padre = miArchivo.getParent();
 JOptionPane.showMessageDialog(null,
 "El padre es: " + padre);

 String absoluta = miArchivo.getAbsolutePath();
 JOptionPane.showMessageDialog(null,
 "La ruta absoluta es: " + absoluta);
 boolean estáAhí = miArchivo.exists();

 String nombre = miArchivo.getName();
 JOptionPane.showMessageDialog(null,
 "El nombre es: " + nombre);

 boolean verificarDirectorio = miArchivo.isDirectory();
 long miLongitud = miArchivo.length();
 String [] todosLosArchivos = miArchivo.list();
}
```

y a continuación veremos la explicación de los métodos de la clase **File** que se utilizaron.

Al principio, para construir una instancia de la clase **File** proporcionamos una cadena de texto. Una vez creada esta instancia, podemos usarla. He aquí la instancia:

```
File miArchivo = new File("c:\\temp\\java\\demo.txt");
```

Como alternativa podríamos usar un campo de texto como antes o un selector de archivos, el cual permite que el usuario explore los directorios y haga clic en un archivo. El selector de archivos devuelve una instancia de `File` al programa. Más adelante hablaremos sobre esta herramienta.

Cabe mencionar que, por casualidad, el carácter de escape \ de Java es igual que el separador de archivos de Windows. En este caso necesitamos decir que la barra diagonal inversa es simplemente un carácter normal de la cadena en vez de ser uno especial, y por ende lo escapamos. Efectivamente, cuando usamos el nombre de un archivo entre comillas en un programa, \\ representa a un \ ordinario.

### **getAbsolutePath**

Para encontrar la ruta absoluta de una instancia de `File` usamos lo siguiente:

```
String absoluta = miArchivo.getAbsolutePath();
```

y obtenemos:

```
c:\\temp\\java\\demo.txt
```

### **getName**

Para extraer el nombre del archivo usamos:

```
String nombre = miArchivo.getName();
```

El valor devuelto es `demo.txt`.

### **getParent**

Para encontrar el directorio padre (el que contiene la instancia de `File` en cuestión) usamos:

```
String padre = miArchivo.getParent();
```

y obtenemos el siguiente resultado:

```
c:\\temp\\java
```

### **exists**

Para comprobar que el archivo existe, podemos usar:

```
boolean estáAhí = miArchivo.exists();
```

y obtenemos `true` en este caso.

### **isDirectory**

El archivo podría ser un directorio. Para verificar esto usamos:

```
boolean verificarDirectorio = miArchivo.isDirectory();
```

lo cual devuelve `true` si el archivo es un directorio y `false` en caso contrario.

**length**

Podemos encontrar el tamaño del archivo en bytes. Cabe mencionar que debemos convertir el valor devuelto en un **int**:

```
int miLongitud = (int)miArchivo.length();
```

**list**

Podemos llenar un arreglo de cadenas con una lista de nombres de archivos dentro de un directorio:

```
String[] todosLosArchivos = miArchivo.list();
```

En el ejemplo actual nuestro objeto **File** (nos referimos a “**demo.txt**”) no es un directorio, por lo que el método **list** devuelve **null**.

## ● La clase JFileChooser

Cuando abrimos un archivo mediante un procesador de palabras o un editor, por lo general aparece un cuadro de diálogo que nos permite explorar las carpetas y seleccionar un archivo. Este componente se incluye en Swing por medio de la clase **JFileChooser**. Esta clase tiene dos variaciones, para abrir archivos y guardarlos.

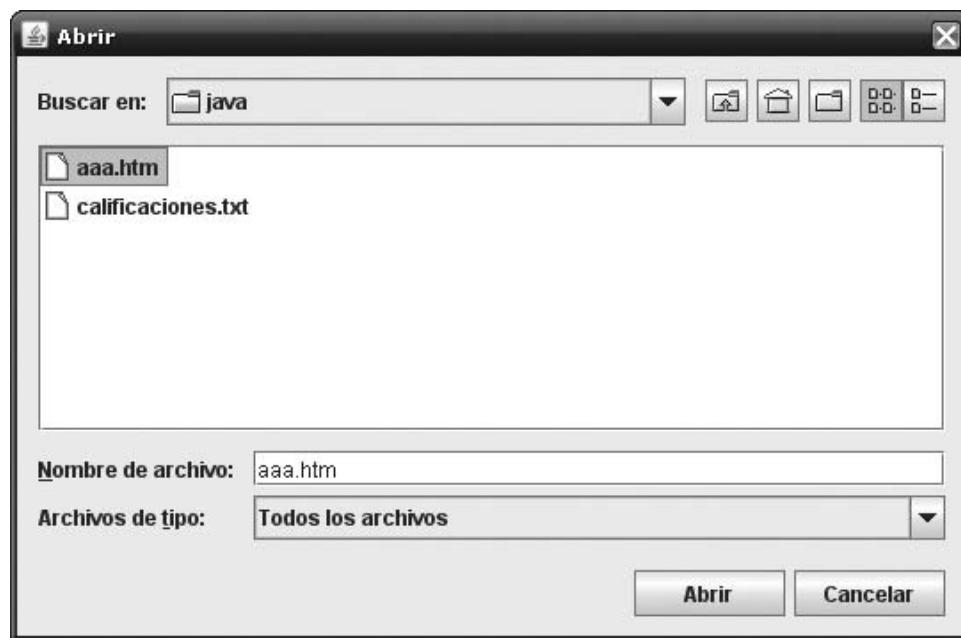
He aquí un programa que muestra las principales características del selector de archivos. La interfaz de usuario consiste simplemente en un botón “abrir” y un botón “guardar”, junto con un campo de texto para mostrar el nombre del archivo seleccionado. La figura 17.4(a) muestra la interfaz **DemoFileChooser** y la figura 17.4(b) muestra el selector de archivos en acción, cuando el usuario hace clic en el botón “abrir”.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.io.*;

public class DemoFileChooser extends JFrame
 implements ActionListener {
```



**Figura 17.4(a)** Pantalla del programa **DemoFileChooser**.



**Figura 17.4(b)** Pantalla del selector de archivos que aparece al hacer clic en el botón “abrir” de `DemoFileChooser`.

```
private JButton botónAbrir, botónGuardar;
private JFileChooser selectorArchivos;
private JTextField campoNombre;

public static void main(String [] args) {
 DemoFileChooser marco = new DemoFileChooser();
 marco.setSize(300, 150);
 marco.crearGUI();
 marco.setVisible(true);
}

private void crearGUI() {
 setDefaultCloseOperation(EXIT_ON_CLOSE);
 Container ventana = getContentPane();
 ventana.setLayout(new FlowLayout());

 botónAbrir = new JButton("abrir");
 ventana.add(botónAbrir);
 botónAbrir.addActionListener(this);

 botónGuardar = new JButton("guardar");
 ventana.add(botónGuardar);
 botónGuardar.addActionListener(this);
```

```

campoNombre = new JTextField(25);
ventana.add(campoNombre);
}

public void actionPerformed(ActionEvent event) {
 File archivoSeleccionado;
 int respuesta;
 if (event.getSource() == botónGuardar) {
 selectorArchivos = new JFileChooser();
 respuesta = selectorArchivos.showSaveDialog(this);
 if (respuesta == JFileChooser.APPROVE_OPTION) {
 archivoSeleccionado = selectorArchivos.getSelectedFile();
 campoNombre.setText(archivoSeleccionado.getAbsolutePath());
 }
 }
 if (event.getSource() == botónAbrir) {
 selectorArchivos = new JFileChooser();
 respuesta = selectorArchivos.showOpenDialog(this);
 if (respuesta == JFileChooser.APPROVE_OPTION) {
 archivoSeleccionado = selectorArchivos.getSelectedFile();
 campoNombre.setText(archivoSeleccionado.getAbsolutePath());
 }
 }
}
}

```



Es importante observar que el componente selector de archivos no transfiere ningún dato a los archivos. El programador tiene que codificar esto, como vimos en los ejemplos anteriores. Todo lo que el selector de archivos hace es proveer al programa los detalles sobre el archivo que el usuario seleccionó. Esto es mucho más conveniente que usar campos de texto, ya que se reducen los posibles errores del usuario, como seleccionar un archivo que no exista.

He aquí las principales características de un selector de archivos:

- Creamos una instancia de la clase **JFileChooser**.
- Para mostrar el selector de archivos usamos el método **showOpenDialog** o **showSaveDialog**. Por lo general, lo mostramos centrado en el marco actual, para lo cual proporcionamos el parámetro **this**.
- El selector de archivos devuelve un código numérico que indica si el usuario canceló el cuadro de diálogo o no. Podemos comparar este código con la constante **JFileChooser.APPROVE\_OPTION** para averiguar si se seleccionó un archivo.
- El método **getSelectedFile** devuelve una instancia de la clase **File**. Para obtener la ruta del archivo como una cadena de texto utilizamos el método **getAbsolutePath** de la clase **File**.
- Se puede usar la misma instancia de **JFileChooser** como un cuadro de diálogo “guardar” o “abrir”.

He aquí el fragmento de código que crea un diálogo para guardar:

```
selectorArchivos = new JFileChooser();
respuesta = selectorArchivos.showSaveDialog(this);
if(respuesta == JFileChooser.APPROVE_OPTION) {
 archivoSeleccionado = selectorArchivos.getSelectedFile();
 campoNombre.setText(archivoSeleccionado.getAbsolutePath());
}
```

## PRÁCTICA DE AUTOEVALUACIÓN

- 17.5** Ejecute el programa `DemoFileChooser`. ¿Cuál es la diferencia entre la versión “guardar” y la versión “abrir”?

## E/S de consola

En la historia de la computación han evolucionado tres estilos de interfaz de usuario. En un principio se utilizó el método de la línea de comandos, en el cual se mostraba un indicador en la pantalla, consola o “terminal” (al final de un cable) y el usuario escribía datos o un comando después del indicador, en la línea de comandos. Después la pantalla se “desplazaba hacia arriba” y aparecía el siguiente indicador. La entrada y la salida eran puramente textuales y sólo había un lugar en donde podía ocurrir la entrada. El siguiente progreso fue el uso de los menús, en donde los usuarios podían mover un cursor sobre un menú y oprimir teclas para introducir datos o seleccionar opciones. Poco después los sistemas de ventanas agregaron un ratón como dispositivo de señalamiento y fue posible tener varias aplicaciones en pantalla a la vez.

De hecho, podemos usar el lenguaje Java para escribir cualquiera de estos estilos de programa y aquí veremos las herramientas para el software de línea de comandos. Pero ¿por qué queríamos crear dicho software? Bueno, los sistemas operativos Unix y GNU/Linux están basados en esta metodología; además tienen una extensa colección de programas utilitarios: herramientas de software. El poder proviene en parte de la capacidad de crear nuevos programas mediante el proceso de unir las herramientas existentes. Por lo general, cada herramienta es una aplicación independiente que acepta cierto tipo de entrada textual, la procesa de alguna forma y pasa el texto modificado a otra herramienta. Dicho software no necesita usar las clases de Swing para nada. Sólo necesitamos la capacidad de desplegar mensajes en la pantalla e introducir datos o comandos. Para ello podemos usar una simple combinación de flujos y la clase `System`.

## La clase System

Esta clase provee tres flujos listos para usar:

```
System.out
System.in
System.err
```

Podemos usar estos flujos con una ventana desplazable conocida como pantalla de E/S de “terminal” o “consola”.

El flujo `System.out` provee el método `print`, que imprime una cadena, o `println`, que imprime una cadena seguida del código `\n` de fin de línea. Recuerde que también podemos usar `+` para unir cadenas antes de imprimirlas. El término *imprimir* se heredó de los días de las impresiones en papel. He aquí un programa completo que imprime cierto texto como salida:

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

La clase `System` se importa en forma automática y no se requiere el manejo de excepciones para la salida. Cabe mencionar que la cadena a imprimir también puede contener varias líneas de texto. Este flujo es el equivalente directo de `System.in`, en cuanto a que su salida se puede redirigir o canalizar al nivel del sistema operativo. No necesitamos crear este flujo con `new`; ya existe de antemano.

Podemos usar el flujo `System.in` para la entrada directa desde el teclado; este flujo también puede recibir texto que se canalice o redirija hacia él por medio de un comando del sistema operativo. Sin embargo, los métodos que se proporcionan con este flujo son de un nivel tan bajo que el programador de Java, acostumbrado a la conveniencia de las cadenas de texto, no podría utilizarlos. Recomendamos la siguiente metodología para la E/S de consola, la cual nos provee un objeto `BufferedReader` y la capacidad de usar nuestro ya conocido método `readLine`. Para crear el flujo podemos usar lo siguiente:

```
private BufferedReader teclado;
teclado = new BufferedReader(
 new InputStreamReader(System.in));
```

Usamos `BufferedReader` para leer de archivos en nuestros ejemplos anteriores. Aquí lo usaremos con un objeto `InputStreamReader` que trabaja sobre `System.in`.

Una vez creado el flujo, para leer datos de éste hacemos lo siguiente:

```
String linea = teclado.readLine();
```

En las clases `Buscador` y `MiniNavegador` que veremos más adelante hay un método llamado `indicador`, el cual utiliza la entrada y salida desde el teclado. Los puntos principales son:

- Para usarlo se proporciona un mensaje a mostrar en pantalla y devuelve la cadena que el usuario escribió. Por ejemplo:

```
String respuesta = indicador("escriba su nombre:");
```

- El método de búfer para la salida de consola es ligeramente distinto al de los archivos. Usamos la siguiente instrucción:

```
System.out.flush();
```

La cual asegura que cualquier texto enviado con `print` se muestre inmediatamente, en vez de mostrarse hasta que ocurra el siguiente `println`.

- El uso del flujo de entrada involucra una excepción verificada. Esto es muy poco probable que ocurra, por lo que la manejamos en forma local en vez de complicar las cosas para el método que llama a `indicador`.

El flujo `System.err` se puede utilizar de manera similar con `print` y `println`, sólo que no podemos redirigir ni canalizar su salida; no interfiere con la salida real del programa. He aquí un ejemplo:

```
System.err.println("Error... el programa va a terminar.");
```

La clase **System** también provee un método para terminar las aplicaciones de inmediato, el cual se conoce como **System.exit**. También debemos suministrar un código entero, que el sistema operativo puede usar para determinar la causa de terminación. La convención es que cero indica que todo está bien, mientras que un código distinto de cero indica que algo salió mal. Usted puede elegir cuáles códigos de error distintos de cero desea utilizar; debe hacer las observaciones correspondientes en la documentación para su aplicación. He aquí algunos ejemplos:

```
if (edad >= 0){
 System.exit(0); // salida normal
}
else {
 System.err.println("Error en el programa");
 System.exit(3); // salida por error
}
```

## ● Uso de JOptionPane

Como alternativa para los flujos de entrada y salida de **System**, podemos utilizar las herramientas de **JOptionPane** de la biblioteca Swing. No hay que crear una ventana gráfica, como con las aplicaciones Swing completas. He aquí un ejemplo:

```
import javax.swing.*;
public class DemoJOptionPane{
 public static void main(String[] args){
 JOptionPane.showMessageDialog(null, "Hola mundo");
 }
}
```

Observe que es necesario importar las bibliotecas de Swing.

## ● Un ejemplo de consola: Buscador

Ahora vamos a utilizar los flujos de **System**. Escribiremos un programa que solicita el nombre de un archivo y una subcadena. Después muestra cada línea del archivo que contiene la subcadena.

Como los archivos pueden contener muchas líneas similares, vamos a proveer también un contexto; mostraremos la línea anterior y la siguiente.

Entonces, nuestro programa tiene dos entradas de consola: el archivo a buscar y la subcadena requerida. Mostramos indicadores para ambas entradas y las leemos de **System.in**. Más adelante veremos el método de los “argumentos de la línea de comandos” para obtener datos en los progra-

mas de consola. Una vez abierto el archivo, utilizamos el método `indexOf` para buscar una subcadena; el método devuelve `-1` si la subcadena no está presente. El método `indicador` ilustra la E/S desde el teclado. He aquí el programa:

```
import java.io.*;

public class Buscador {
 private String linea1, linea2, linea3;
 private BufferedReader teclado;
 private BufferedReader flujoEnt;

 public static void main (String [] args) {
 Buscador unHallazgo = new Buscador();
 unHallazgo.hacerBúsqueda();
 }

 private void hacerBúsqueda() {
 teclado = new BufferedReader(
 new InputStreamReader(System.in));
 String nombreArchivo = indicador("Escriba el archivo a buscar: ");
 String seBusca = indicador("Escriba la cadena a buscar: ");
 linea1 = "";
 linea2 = "";
 try {
 flujoEnt = new BufferedReader(new
 FileReader(nombreArchivo));
 while ((linea3 = flujoEnt.readLine()) != null) {
 if (linea2.indexOf(seBusca) >= 0) {
 mostrarLínea();
 }
 // avanza al siguiente grupo de 3
 linea1 = linea2;
 linea2 = linea3;
 // y obtiene la nueva linea3 del archivo...
 }
 // revisa la última línea
 linea3 = ""; //elimina valor null fin archivo (eof)
 if (linea2.indexOf(seBusca) >= 0) {
 mostrarLínea();
 }
 flujoEnt.close();
 }
 catch (IOException e) {
 System.err.println("Error en Buscador: "+ e.toString());
 System.exit(1);
 }
 }

 private void mostrarLínea() {
 System.out.println("[== contexto:");
 System.out.println(linea1);
```

```

 System.out.println(línea2);
 System.out.println(línea3);
 System.out.println("===");
 System.out.println("");
 }

 private String indicador(String mensaje) {
 String respuesta = "";
 try {
 System.out.print(mensaje);
 System.out.flush();
 respuesta = teclado.readLine();
 }
 catch (IOException e) {
 System.err.println("Teclado " + e.toString());
 System.exit(2);
 }
 return respuesta;
 }
}

```



La figura 17.5 muestra lo que obtenemos al buscar la cadena `if` en el archivo `Buscador.java`:

```

Escriba el archivo a buscar: c:\temp\java\Buscador.java
Escriba la cadena a buscar: if
[== contexto:
 while ((línea3 = flujoEnt.readLine()) != null) {
 if (línea2.indexOf(seBusca) >= 0) {
 mostrarLínea()
 ===]
[== contexto:
 línea3 = ""; //elimina valor null fin archivo (eof)
 if (línea2.indexOf(seBusca) >= 0) {
 mostrarLínea()
 ===]

```

**Figura 17.5** Salida del programa `Buscador`.

### Cómo leer desde un sitio remoto

Sorprendentemente, leer una página Web de un servidor remoto no es más difícil en Java que leer un archivo local; esto se debe al poder de la biblioteca de Java. En vez de un nombre de archivo, necesitamos proveer la dirección Web (URL – Localizador Uniforme de Recursos) del archivo. A continuación veremos una aplicación de consola que pide al usuario un URL y que muestra el

```
Escriba un URL (ej: http://java.sun.com/): http://java.sun.com

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"><html xmlns="http://www.w3.org/1999/xhtml"><head><meta http-equiv="Content-Type" content="text/html; charset=utf-8" /><script type="text/javascript"> var _U = "undefined"; var g_HttpRelativeWebRoot = "/ocom/"; var SSContributor = false; var SSForceContributor = false; var SSHideContributorUI = false; var ssUrlPrefix = "/technetwork/"; var ssUrlType = "2"; var g_navNode_Path = new Array(); g_navNode_Path[0] = 'otnen'; g_navNode_Path[1] = 'otnen_java'; var g_ssSourceNodeId = "otnen_jAVA"; var g_ssSourceSiteId = "otnen"; var g_strLanguageId = "en";</script><script id="SSNavigationFunctionsScript" type="text/javascript" src="/ocom/website/s/otnen/sitenavigationfunctions.js"></script>
```

Figura 17.6 Salida del programa MiniNavegador.

contenido del archivo en la pantalla. El programa sólo puede mostrar texto, como podemos ver en la pantalla de consola de la figura 17.6.

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

public class MiniNavegador {
 private BufferedReader flujoEnt, teclado;
 public static void main (String [] args) {
 MiniNavegador unNavegador = new MiniNavegador();
 unNavegador.obtener();
 }

 private void obtener() {
 String cadenaUrl = "";
 String linea;
 teclado = new BufferedReader(new
 InputStreamReader(System.in));
 try {
 cadenaUrl = indicador("Escriba un URL " +
 "(ej: http://java.sun.com/): ");
 // crea una conexión a un URL
 URL direcciónUrl = new URL(cadenaUrl);
 URLConnection conexión =
 direcciónUrl.openConnection();
 flujoEnt = new BufferedReader(new
 InputStreamReader(conexión.getInputStream()));
 while ((línea = flujoEnt.readLine()) != null) {
 System.out.print(línea);
 }
 }
 catch (MalformedURLException e) {
 System.err.println(cadenaUrl + e.toString());
 System.exit(2);
 }
 }
}
```

```

 catch (IOException e) {
 System.err.println("Error al acceder a URL: " +
 e.toString());
 System.exit(1);
 }
 }

private String indicador(String mensaje) {
 String respuesta = "";
 try {
 System.out.print(mensaje);
 System.out.flush();
 respuesta = teclado.readLine();
 }
 catch (IOException e) {
 System.err.println("Teclado " + e.toString());
 System.exit(2);
 }
 return respuesta;
}
}

```

La mayor parte del programa se enfoca en el manejo de excepciones y la E/S de consola. El código esencial para el URL es:

```

import java.net.*;
// crea una conexión a un URL
URL direcciónUrl = new URL(cadenaUrl);
URLConnection conexión =
 direcciónUrl.openConnection();
flujoEnt = new BufferedReader(new
 InputStreamReader(conexión.getInputStream()));

```

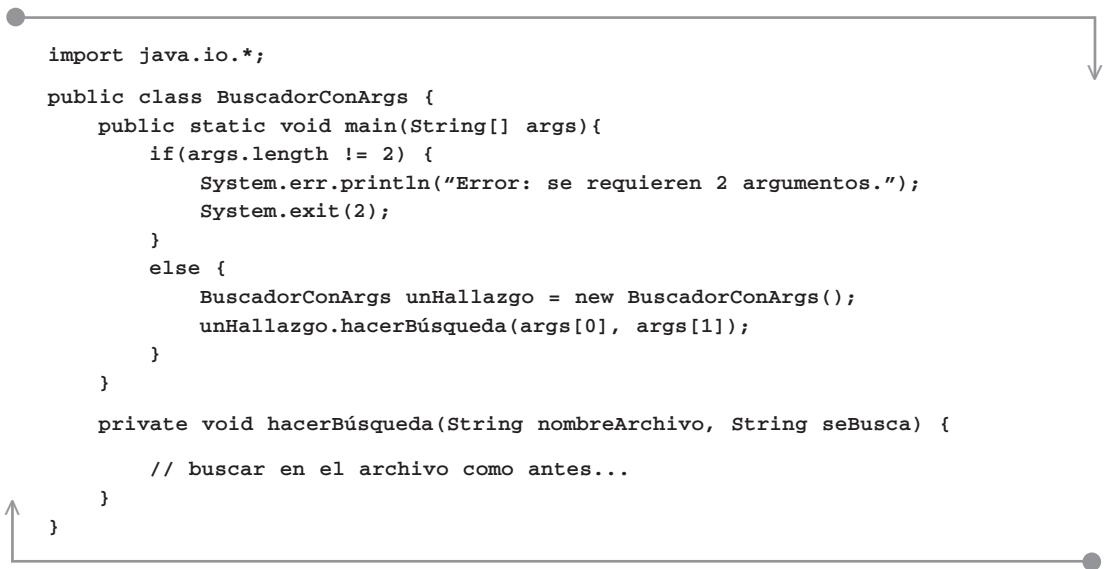
Primero proveemos una cadena a la clase `URL`, la cual efectúa varias verificaciones en relación con la sintaxis (como el uso correcto de los caracteres / y .). Si hay un error se lanza una excepción `MalformedURLException`.

Después creamos una conexión con la clase `URLConnection`.

Como podemos ver de la figura 17.6, éste es un navegador bastante primitivo; no puede interpretar etiquetas de HTML (encerradas entre paréntesis angulares). Todo lo que podemos ver es el HTML “puro”.

## ● Argumentos de la línea de comandos

Los argumentos de la línea de comandos proveen otra forma de introducir valores iniciales (como nombres de archivo) en los programas de consola. Para usarlos necesitamos ejecutar el programa desde la línea de comandos, como se describe en el apéndice I. Recuerde nuestro programa `Buscador`. Vamos a replantearlo de manera que el archivo a usar y la cadena a buscar se pasen desde la línea de comandos. He aquí parte del código. La parte que se omite es idéntica a la de `Buscador`.



```
import java.io.*;

public class BuscadorConArgs {
 public static void main(String[] args) {
 if(args.length != 2) {
 System.err.println("Error: se requieren 2 argumentos.");
 System.exit(2);
 }
 else {
 BuscadorConArgs unHallazgo = new BuscadorConArgs();
 unHallazgo.hacerBúsqueda(args[0], args[1]);
 }
 }

 private void hacerBúsqueda(String nombreArchivo, String seBusca) {
 // buscar en el archivo como antes...
 }
}
```

He aquí un ejemplo de cómo ejecutar el programa:

```
java BuscadorConArgs "c:\temp\miarchivo.txt" "mis intereses"
```

Cabe mencionar que cuando utilizamos un IDE para desarrollar programas de línea de comandos, por lo general hay una herramienta para preestablecer los argumentos de la línea de comandos. Esto es conveniente, pero a la hora de desarrollar completamente el programa y usarlo de verdad, lo ejecutaremos sin el IDE y los argumentos se introducen en la línea de comandos como en el ejemplo anterior.

Éstos son los puntos principales:

- La frase “argumentos de línea de comandos” es tradicional. En términos de Java, un argumento es un parámetro por lo que nos apegaremos al término “parámetro” de aquí en adelante.
- El método `main` del programa tiene una matriz de cadenas como parámetro. En ejemplos anteriores no lo habíamos utilizado.
- La matriz de parámetros se llama `args`. Podríamos elegir nuestro propio nombre, pero es una tradición utilizar `args`.
- Cuando el programa se empieza a ejecutar, se hace una llamada a `main` como siempre. Además, los elementos que vayan después del nombre del programa en la línea de comandos se transfieren a la matriz `args`. El primer elemento va en `args[0]`, el segundo en `args[1]`, etc.
- En la línea de comandos, los elementos se separan por uno o varios espacios. Si un elemento en sí contiene un espacio, puede ir encerrado entre comillas dobles. En la línea de comandos anterior optamos por encerrar ambos elementos entre comillas. Las comillas se eliminan antes de colocar los elementos en la matriz `args`.
- Para averiguar el número de parámetros podemos usar `length`. Es una buena práctica comprobar el número correcto de parámetros, como se muestra en el código anterior.

El uso de argumentos de la línea de comandos puede simplificar el uso de un programa, pero no es algo adecuado para los programas interactivos en los que deseamos verificar la entrada de un usuario y solicitar que corrija esa entrada en caso de errores.

## Fundamentos de programación

- Los programas usan flujos para leer datos de los archivos y escribir datos en ellos.
- Utilizamos archivos para preservar los datos después de ejecutar un programa, o para pasar datos a otros programas.

## Errores comunes de programación

- Para acceder a un archivo debemos usar `try-catch` para una excepción `IOException` (lo cual es preferible) o debemos declarar que un método que usa flujos lanza una excepción `IOException`.
- No es posible usar la siguiente instrucción:

```
miArchivo = new PrintWriter("c:\\\\temp\\\\demo.txt");
```

En vez de ello debemos usar:

```
miArchivo = new PrintWriter(
 new FileWriter("c:\\\\temp\\\\demo.txt"), true);
```

o, para la entrada:

```
miArchivo = new BufferedReader(
 new FileReader("c:\\\\temp\\\\demo.txt"));
```

- Para producir un \ entre comillas, debemos usar \\.
- El método `System.exit` requiere un parámetro entero.

## Secretos de codificación

No hemos visto nuevas instrucciones ni herramientas del lenguaje, ya que las herramientas de archivos las proporcionan las clases en vez de estar integradas en el lenguaje.

## Nuevos elementos del lenguaje

Introdujimos estas clases:

```
PrintWriter
BufferedReader
URL
URLConnection
File
JFileChooser
```

## Resumen

- Los archivos se abren, después se escribe en ellos (o se lee de ellos) y por último se cierran.
- Para declarar y crear flujos de archivos podemos usar el siguiente ejemplo:

```
private PrintWriter flujoSalida;
flujoSalida = new PrintWriter(
 new FileWriter("c:\\temp\\demo.txt"), true);

private FileReader flujoEntrada;
flujoEntrada = new BufferedReader(
 new FileReader("c:\\temp\\cualquiera.txt"));
```

- El programador es libre de elegir los nombres de las variables de flujo. Estos nombres se asocian con los nombres de archivos del sistema operativo actual.
- El método `readLine` es un método conveniente para acceder a los flujos de entrada.
- Los métodos `print` y `println` son convenientes para los flujos de salida.
- El estilo popular de leer un archivo línea por línea es:

```
String linea;
while((linea = flujoEntrada.readLine()) != null) {
 // procesar linea
}
```

- Podemos cerrar flujos de la siguiente manera:

```
flujoEntrada.close();
```

- La clase `JFileChooser` provee cuadros de diálogo que permiten al usuario explorar directorios y hacer clic en el nombre de un archivo para seleccionarlo.
- La clase `System` provee flujos ya creados:

```
System.in
System.out
System.err
```

- El método `System.exit(n)` se puede usar para abandonar una aplicación de inmediato.
- Los argumentos (parámetros) de línea de comandos se transfieren de la línea de comandos hacia el parámetro tipo matriz `String` del método `main`.

## Ejercicios

El primer conjunto de preguntas involucra aplicaciones Swing.

- 17.1** Escriba un programa de Swing que, al hacer clic en un botón, envíe como salida su nombre y dirección a un archivo especificado. Use un editor para verificar que el archivo tenga el contenido esperado.
- 17.2** Escriba un programa de Swing que cuente el número de líneas en un archivo especificado. Asegúrese de que trabaje con un archivo vacío y que produzca un valor de 0.
- 17.3** Escriba un programa de Swing que pueda leer un archivo que contenga líneas con tres elementos: un nombre (sin espacios) y dos enteros. Los elementos deben estar separados por comas. La aplicación debe permitir que se seleccione el nombre del archivo mediante un campo de texto; además debe proveer un botón “siguiente” para que se introduzca la siguiente línea de entrada del archivo y se muestre en tres campos de texto. Indique al usuario cuando se llegue al final del archivo.
- 17.4**
  - (a) Escriba un programa de Swing que sea un editor de texto simple. En un principio utilice un campo de texto para los nombres. Debe tener lo siguiente:
    - Un campo de texto para aceptar el nombre del archivo.
    - Un área de texto para edición.
    - Un botón “abrir” para introducir el texto del archivo seleccionado.
    - Un botón “guardar” para guardar el contenido del área de texto en el archivo seleccionado.
    - Un botón “salir”.
  - (b) Modifique su editor para utilizar un selector de archivo en vez de un campo de texto.
  - (c) Incorpore menús en vez de botones, como se describe en el apéndice A.
- 17.5** Escriba un programa de Swing que lea dos archivos línea por línea y compare si los archivos son iguales. Debe mostrar un mensaje para indicar que los archivos son o no iguales. Use selectores de archivos para obtener los nombres de los archivos.
- 17.6** Escriba un programa de Swing que reemplace una cadena por otra en todo un archivo; escriba la nueva versión en otro archivo. Use selectores de archivos para obtener los nombres de los archivos y use dos campos de texto para aceptar las cadenas “de” y “para”. Utilice el método `replace` del capítulo 15.

Las siguientes preguntas están relacionadas con aplicaciones de consola.

- 17.7** Escriba un programa de consola que compare dos archivos línea por línea. Debe imprimir un mensaje para indicar que los archivos son o no iguales. Use cuadros de diálogo para que el usuario introduzca los nombres de los archivos.
- 17.8** Escriba un programa de consola que reemplace una subcadena por otra en un archivo; escriba la nueva versión en otro archivo. La aplicación debe usar argumentos de línea de comandos para los nombres del archivo anterior y el archivo nuevo, además de las dos subcademas.
- 17.9** Escriba un programa de consola para generar “listas de directorio”. El programa debe pedir al usuario una ruta absoluta para después mostrar todos los nombres de los archivos en ese directorio. Si alguno de los archivos es directorio, se debe imprimir la palabra “dir” después de su nombre.

- 17.10** Modifique el programa **MiniNavegador** de maneara que repita el indicador si se produce una excepción **MalformedURLException**. Use un bloque **try-catch** dentro de un **while**.
- 17.11** Modifique el programa **MiniNavegador** para que en vez de imprimir el HTML en la pantalla, lo envíe a un archivo. Se debe pedir al usuario el nombre del archivo. Para asegurar que el nombre del archivo se escriba en forma apropiada, véalo con un navegador Web.

## Respuestas a las prácticas de autoevaluación

- 17.1** **PrintWriter** no tiene un constructor con una cadena que contenga un nombre de archivo como su parámetro. En vez de ello debemos hacer lo siguiente:

```
private PrintWriter archivoSalida;
...
archivoSalida = new PrintWriter(
 new FileWriter(campoNombre.getText()), true);
```

- 17.2** El área de texto terminaría con una línea extensa. Al introducir una cadena desde un archivo se elimina el marcador de fin de línea, por lo que necesitamos insertar uno al momento de adjuntar la línea.
- 17.3** No hay **readLine** en el ciclo, por lo que éste no avanzará por el archivo. El primer valor de **línea** se adjuntará al área de texto una y otra vez.
- 17.4** Modifique el siguiente código:

```
if (campoPersona.getText().equals(nombreEnArchivo)) {
 encontró = true;
```

de manera que se convierta en:

```
if (nombreEnArchivo.equalsIgnoreCase(campoPersona.getText())) {
 encontró = true;
}
```

- 17.5** La versión “abrir” tiene una barra de título y un botón identificado como “Abrir”, mientras que la versión “guardar” usa un botón identificado como “Guardar”. Su funcionalidad es similar.

# CAPÍTULO 18



## Diseño orientado a objetos

En este capítulo conoceremos cómo:

- Identificar las clases que se necesitan en un programa.
- Diferenciar entre la composición y la herencia.
- Utilizar algunos lineamientos para el diseño de clases.

### ● Introducción

Probablemente para empezar a diseñar un puente no necesite pensar en el tamaño de los remaches. Primero habría que tomar decisiones importantes, como si el puente debe ser voladizo o suspendido. Para empezar a diseñar un edificio no pensaría en el color de las alfombras. Primero tomaría decisiones importantes, como cuántos pisos debe haber y en dónde deben estar los elevadores.

Para un programa pequeño en realidad no se necesita el diseño; podemos simplemente crear la interfaz de usuario y pasar inmediatamente a escribir las instrucciones de Java. Pero es muy sabido que para los programas extensos el programador debe empezar con las decisiones importantes en vez de con los detalles. Esto significa pensar en las clases que se necesitan. El programador debe hacer el diseño, hacerlo primero y hacerlo bien. Tiene que posponer las decisiones sobre los detalles (como el formato exacto de un número o la posición de un botón). Desde luego, todas las etapas de la programación son cruciales, pero algunas lo son más que otras.

Cuando empezamos a escribir programas, por lo general invertimos mucho tiempo en la prueba y el error. A menudo esto es muy divertido y creativo. Algunas veces pasamos cierto tiempo luchando con el lenguaje de programación. Se requiere tiempo para aprender las buenas prácticas y reconocer las malas. Se requiere aún más tiempo para adoptar una metodología de diseño efectiva

para la programación. La diversión y la creatividad permanecen, al tiempo que se reducen las partes molestas de la programación.

El proceso de diseño recibe como entrada la especificación de lo que el programa debe hacer. El producto final del proceso de diseño es una descripción de las clases y los métodos que empleará el programa.

En este capítulo le explicaremos cómo usar una metodología ampliamente conocida para el diseño de programas OO. Utilizaremos el ejemplo sencillo del programa del globo para ilustrar cómo hacer el diseño. También presentaremos un ejemplo de diseño más complejo.

## ● El problema del diseño

Hemos visto con anterioridad que un programa OO consiste en una colección de objetos. El problema al empezar a desarrollar un nuevo programa es identificar los objetos apropiados. Una vez que los hayamos identificado, cosecharemos todos los beneficios de la POO. Pero el problema fundamental de la POO es cómo identificar los objetos. Esto es lo que ofrece un método de diseño: una metodología o serie de pasos para identificar los objetos. Es justo igual que cualquier otro tipo de diseño: necesitamos un método. No basta con conocer los fundamentos de la POO. Como analogía, conocer las leyes de la física no significa que podamos diseñar una nave espacial; también tenemos que llevar a cabo cierto proceso de diseño.

Uno de los principios utilizados en el diseño de los programas OO es simular las situaciones del mundo real como objetos. Construimos un modelo de software de las cosas que hay en el mundo real. He aquí algunos ejemplos:

- Si vamos a desarrollar un sistema de automatización de oficinas, debemos simular a los usuarios, el correo, los documentos compartidos y los archivos.
- En un sistema de automatización industrial debemos simular las distintas máquinas, las colas de trabajo, los pedidos y las entregas.

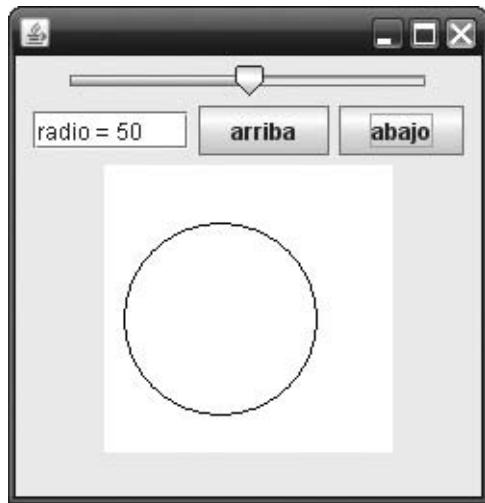
La metodología es identificar los objetos en el problema que buscamos resolver y modelarlos como objetos en el programa.

La abstracción juega un papel importante en este proceso. Sólo necesitamos modelar las partes relevantes del problema que se tiene que resolver, por lo cual podemos ignorar cualquier detalle irrelevante. Si modelamos un globo, necesitamos representar su posición, su tamaño y su color. Pero no necesitamos modelar el material del que está hecho. Si vamos a crear un sistema de registros de personal, probablemente tengamos que modelar los nombres, las direcciones y las descripciones de los puestos, pero no los pasatiempos ni los estilos de música preferidos.

## ● Cómo identificar los objetos y métodos

Una manera efectiva de llevar a cabo el diseño OO es examinar la especificación de software para extraer información sobre los objetos y métodos. La metodología para identificar los objetos y métodos es:

1. Buscar sustantivos (cosas) en la especificación; éstos son los objetos.
2. Buscar verbos (palabras de acción) en la especificación; éstos son los métodos.



**Figura 18.1** El programa del globo.

Por ejemplo, a continuación le mostramos la especificación para el sencillo programa del globo:

Escriba un programa para representar un globo y manipularlo mediante una GUI. El globo debe mostrarse como un círculo en un panel. Utilice botones para cambiar la posición del globo y moverlo una distancia fija hacia arriba o hacia abajo. Use un control deslizable para modificar el radio del globo, el cual se debe mostrar en un campo de texto.

La ventana se muestra en la figura 18.1.

Debemos buscar verbos y sustantivos en la especificación. En este caso podemos ver los siguientes sustantivos:

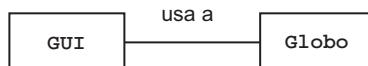
```
GUI, panel, botón, control deslizable, campo de texto, globo, posición,
distancia, radio
```

La GUI provee la interfaz de usuario para el programa. Consiste en botones, un control deslizable, un campo de texto y un panel. La GUI se representa mediante un objeto que es una instancia de la clase **JFrame**. Los objetos botón, control deslizable, campo de texto y panel están disponibles como clases en la biblioteca de Java.

El objeto GUI:

1. Crea los botones, el control deslizable, el campo de texto y el panel en la pantalla.
2. Maneja los eventos de los clics de ratón en los botones y el control deslizable.
3. Crea cualquier otro objeto que se necesite, como el objeto **globo**.
4. Llama a los métodos del objeto **globo**.

El siguiente objeto importante es el globo, que utiliza información para representar su posición (coordenadas *x* y *y*), la distancia que se mueve y su radio. Una opción sería crear varios objetos completos totalmente distintos para representar estos elementos. Pero es más simple representarlos como variables **int**.



**Figura 18.2** Diagrama de clases que muestra las dos clases principales en el programa del globo.

Con esto completamos la identificación de los objetos dentro del programa. Ahora generalizaremos los objetos y las clases de diseño que corresponden a cada objeto. Por lo tanto, necesitamos las clases **GUI**, **JTextField**, **Globo**, etc.

Ahora vamos a extraer los verbos de la especificación:

**cambiarRadio, moverArriba, moverAbajo, mostrarGlobo, mostrarRadio**

Debemos crear los métodos correspondientes dentro del programa que estamos diseñando. Pero necesitamos decidir a cuál objeto pertenecen: al objeto **GUI** o al objeto **Globo**. Parece razonable que los verbos **moverArriba**, **moverAbajo** y **mostrarGlobo** sean métodos asociados con el objeto **Globo**.

Ahora debemos fijar nuestra atención en los verbos **cambiarRadio** y **mostrarRadio**. Ya hemos decidido que el valor del radio se implementará como una variable **int** dentro del objeto **Globo**. Sin embargo, el objeto **GUI** necesita acceso a este valor para mostrarlo en el campo de texto. También necesita cambiar el valor en respuesta a los cambios en el valor del control deslizable. Por lo tanto, la clase **Globo** necesita proveer acceso al valor del radio, lo cual podemos lograr utilizando métodos (**getRadio** y **setRadio**).

Para resumir, nuestro diseño para este programa consiste en dos clases que no son de biblioteca: **GUI** y **Globo**, las cuales se muestran en un diagrama de clases de UML (figura 18.2). Este diagrama muestra las clases principales del programa y sus interrelaciones. La clase **GUI** utiliza a la clase **Globo** mediante llamadas a sus métodos.

Podemos documentar cada clase mediante diagramas de clases más detallados. En estos diagramas, cada cuadro describe a una sola clase y tiene tres secciones que proporcionan información sobre:

1. El nombre de la clase.
2. Una lista de variables de instancia.
3. Una lista de métodos.

Primero veamos la descripción de la clase **GUI**:

|                               |
|-------------------------------|
| <b>Clase GUI</b>              |
| <b>Variables de instancia</b> |
| <b>botónArriba</b>            |
| <b>botónAbajo</b>             |
| <b>deslizable</b>             |
| <b>campoTexto</b>             |
| <b>panel</b>                  |
| <b>Métodos</b>                |
| <b>botónArribaClick</b>       |
| <b>botónAbajoClick</b>        |
| <b>deslizableChange</b>       |

Y éste es el diagrama de la clase **Globo**:

| Clase Globo            |
|------------------------|
| Variables de instancia |
| x                      |
| y                      |
| radio                  |
| intervaloY             |
| Métodos                |
| moverArriba            |
| moverAbajo             |
| mostrar                |
| getRadio               |
| setRadio               |

Ahora el diseño de este programa está completo. El diseño termina una vez que se especifican todas las clases, los objetos y los métodos. Al diseño no le concierne escribir (codificar) las instrucciones de Java que forman estas clases y métodos. Sin embargo, es natural que el lector sienta curiosidad sobre el código, por lo cual ahora le mostraremos el código para la clase **GUI**:

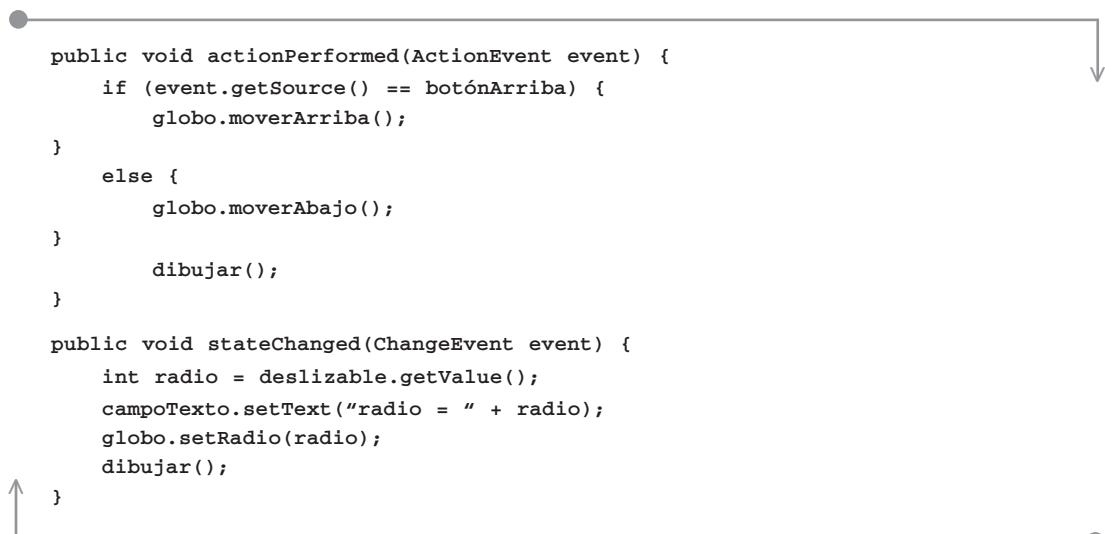
En la parte superior de la clase está la variable de instancia:

```
private Globo globo;
```

Al crear nuevos objetos, necesitamos crear un objeto **Globo**:

```
globo = new Globo();
```

Después tenemos los métodos para responder a los eventos:



```
public void actionPerformed(ActionEvent event) {
 if (event.getSource() == botónArriba) {
 globo.moverArriba();
 }
 else {
 globo.moverAbajo();
 }
 dibujar();
}

public void stateChanged(ChangeEvent event) {
 int radio = deslizable.getValue();
 campoTexto.setText("radio = " + radio);
 globo.setRadio(radio);
 dibujar();
}
```

Y un método compartido:

```
private void dibujar() {
 globo.mostrar(panel);
}
```

A continuación, he aquí el código para la clase **Globo**:

```
import java.awt.*;
import javax.swing.*;

public class Globo {

 private int x = 10;
 private int y = 10;
 private int radio = 20;
 private int intervaloY = 20;

 public void moverArriba() {
 y = y - intervaloY;
 }

 public void moverAbajo() {
 y = y + intervaloY;
 }

 public void mostrar(JFrame panel) {
 Graphics papel = panel.getGraphics();
 papel.setColor(Color.white);
 papel.fillRect(0, 0, 150, 150);
 papel.setColor(Color.black);
 papel.drawOval(x, y, radio * 2, radio * 2);
 }

 public int getRadio() {
 return radio;
 }

 public void setRadio(int nuevoRadio) {
 radio = nuevoRadio;
 }
}
```

Éste es un programa simple, con sólo dos objetos que no son de biblioteca. Sin embargo, ilustra cómo extraer objetos y métodos de una especificación. En breve analizaremos un ejemplo más complejo.

Para sintetizar, el método de diseño para identificar métodos y objetos es:

1. Buscar sustantivos en la especificación; éstos son los objetos (o algunas veces simples variables).
2. Buscar verbos en la especificación; éstos son los métodos.

Una vez que identificamos los objetos, es muy sencillo generalizarlos convirtiéndolos en clases.

Cabe mencionar que aunque este programa consiste en dos clases, podríamos haberlo diseñado como una sola clase. Sin embargo, el diseño que mostramos antes utiliza de manera mucho más explícita los objetos presentes en la especificación del programa. El diseño también separa la parte del programa correspondiente a la GUI de la clase globo. Ésta es una estructura de programa ampliamente recomendada, en donde se separan el código de presentación (GUI) y el modelo (al que algunas veces se le denomina lógica de dominio). A esta estructura se le denomina comúnmente (por razones históricas) como arquitectura modelo-vista-controlador. Esto nos permite modificar un programa con más facilidad, ya que la GUI se puede cambiar de manera independiente a la lógica interna. Por ejemplo, suponga que deseamos agregar otro control deslizable a la GUI para controlar la posición del globo. Sin duda, necesitamos realizar una pequeña modificación a la clase `GUI`, pero no tenemos que modificar la clase `Globo`.

## Ejemplo práctico de diseño

He aquí la especificación para un programa mucho más grande:

### Invasor del ciberespacio

El panel (figura 18.3) muestra al defensor y un extraterrestre. El extraterrestre se desplaza de un lado a otro. Cuando choca con una pared, invierte su dirección. El extraterrestre lanza una bomba al azar que se desplaza verticalmente hacia abajo, pero sólo hay una bomba en cualquier momento dado. Si la bomba le pega al defensor, éste pierde. El defensor se desplaza hacia la izquierda o hacia la derecha, de acuerdo con los movimientos del ratón. Al hacer clic en el ratón, el defensor lanza un láser que se desplaza hacia arriba, pero sólo hay un láser en cualquier momento dado. Si el láser le pega al extraterrestre, el defensor gana.



Figura 18.3 El programa del invasor del ciberespacio.

Recuerde que los principales pasos en el diseño son:

1. Identificar los objetos buscando sustantivos en la especificación.
2. Identificar los métodos buscando verbos en la especificación.

Después de analizar la especificación encontramos los siguientes sustantivos (como podríamos esperar, algunos de estos sustantivos se mencionan más de una vez).

`panel, defensor, extraterrestre, pared, bomba, ratón, láser`

Estos sustantivos corresponden a objetos potenciales, y por ende pueden ser clases dentro del programa. Por lo tanto, traducimos estos sustantivos en los nombres de las clases en el diseño. El sustantivo `panel` se traduce en la clase `JPanel`, disponible en la biblioteca. Los sustantivos `defensor` y `extraterrestre` se traducen en las clases `Defensor` y `Extraterrestre`, respectivamente. El sustantivo `pared` no necesita implementarse como clase, ya que se puede acomodar como un detalle dentro de la clase `Extraterrestre`. El sustantivo `bomba` se traduce en la clase `Bomba`. El sustantivo `ratón` no necesita ser una clase, ya que los eventos de clic del ratón se pueden manejar mediante la clase `JFrame` o la clase `Defensor`. Por último, necesitamos una clase `Láser`. En consecuencia, tenemos la siguiente lista de clases que no son de biblioteca:

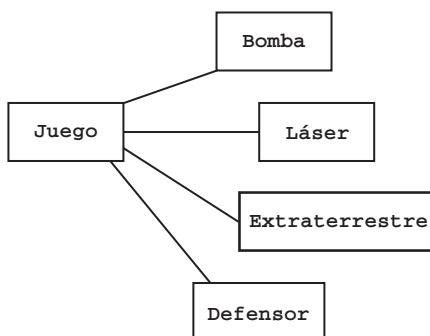
`Juego, Defensor, Extraterrestre, Láser, Bomba`

Estas clases se muestran en el diagrama de clases (figura 18.4). En este diagrama se indica que la clase `Juego` utiliza las clases `Defensor`, `Extraterrestre`, `Láser` y `Bomba`.

Un objeto que hemos ignorado hasta ahora en el diseño es un temporizador que esté configurado para pulsar a pequeños intervalos de tiempo regulares, para poder implementar la animación. Cada vez que el temporizador pulsa, los objetos se desplazan, se borra el contenido del panel y se muestran todos los objetos. Este objeto está disponible como la clase `Timer` en la biblioteca de Java. Otro de los objetos necesarios es el generador de números aleatorios, para controlar cuándo lanza el extraterrestre las bombas. Este objeto está disponible como la clase `Random` en la biblioteca de Java.

Analizamos de nuevo la especificación, esta vez en busca de verbos que podamos adjuntar a la lista anterior de objetos. En esta ocasión podemos ver:

`mostrar, mover, pegar, lanzar, hacer clic, ganar, perder`



**Figura 18.4** Las clases que no son de biblioteca y que están implicadas en el programa del juego.

De nuevo, algunas de estas palabras se mencionan más de una vez. Por ejemplo, tanto los extraterrestres como el defensor se pueden mover. Además, todos los objetos del juego necesitan mostrarse en pantalla.

Ahora tenemos que asignar métodos a las clases, para lo cual nos podemos basar en la especificación.

Podemos documentar cada clase como un diagrama de clases de UML que muestre las variables de instancia y los métodos de cada clase. Empecemos con la clase **Juego**:

|                               |
|-------------------------------|
| <b>clase Juego</b>            |
| <b>Variables de instancia</b> |
| <b>panel</b>                  |
| <b>temporizador</b>           |
| <b>Métodos</b>                |
| <b>mouseMoved</b>             |
| <b>mouseClicked</b>           |
| <b>actionPerformed</b>        |

Ahora vamos a considerar el objeto defensor. Para poder detectar las colisiones, los objetos necesitan saber en dónde están los demás objetos y qué tan grandes son. Éstas son las coordenadas *x* y *y*, junto con la altura y anchura de cada objeto. Podemos representarlas simplemente como variables **int** dentro de las clases **Defensor**, **Extraterrestre**, **Láser** y **Bomba**. Para acceder a sus valores usamos los métodos llamados **getX**, **getY**, **getAltura** y **getAnchura**. Al mover el ratón se mueve un objeto defensor, por lo que la clase **Defensor** debe proveer un método llamado **mover**. También debe proveer un método para poder mostrarlo en pantalla. Por último, la clase **Defensor** necesita un constructor.

Por lo tanto, la clase **Defensor** tiene la siguiente especificación:

|                               |
|-------------------------------|
| <b>clase Defensor</b>         |
| <b>Variables de instancia</b> |
| <b>x</b>                      |
| <b>y</b>                      |
| <b>altura</b>                 |
| <b>anchura</b>                |
| <b>Métodos</b>                |
| <b>Defensor</b>               |
| <b>mover</b>                  |
| <b>mostrar</b>                |
| <b>getX</b>                   |
| <b>getY</b>                   |
| <b>getAltura</b>              |
| <b>getAnchura</b>             |

A continuación diseñamos la clase **Extraterrestre**. El extraterrestre tiene una posición y un tamaño. Cada vez que el reloj pulsa, se mueve. Su dirección y velocidad se controlan mediante el tamaño del intervalo que se utiliza al moverlo. Se puede crear y mostrar en pantalla.

| clase Extraterrestre   |
|------------------------|
| Variables de instancia |
| x                      |
| y                      |
| altura                 |
| anchura                |
| intervaloX             |
| Métodos                |
| Extraterrestre         |
| mover                  |
| mostrar                |
| getX                   |
| getY                   |
| getAltura              |
| getAnchura             |

Ahora veamos un objeto láser. Este objeto tiene una posición y un tamaño. Se crea, se mueve y se muestra en pantalla.

| clase Láser            |
|------------------------|
| Variables de instancia |
| x                      |
| y                      |
| altura                 |
| anchura                |
| intervaloY             |
| Métodos                |
| Láser                  |
| mover                  |
| mostrar                |
| getX                   |
| getY                   |
| getAltura              |
| getAnchura             |

Por último, una bomba es muy similar a un láser. Una diferencia es que una bomba se mueve hacia abajo, mientras que un láser se mueve hacia arriba.

## PRÁCTICA DE AUTOEVALUACIÓN

### 18.1 Escriba el diagrama para la clase `Bomba`.

Ahora tenemos la lista completa de clases, junto con los métodos y las variables de instancia asociados con cada clase. Hemos modelado el juego y diseñamos una estructura para el programa.

Una vez completo el diseño, podemos proceder a escribir el código (pero tenga cuidado; más adelante revisaremos el diseño de las clases `Defensor`, `Extraterrestre`, `Láser` y `Bomba`).

He aquí el código para la clase `Juego`. Empieza con las declaraciones de las variables de instancia de los principales objetos del juego:

```
private Extraterrestre extraterrestre;
private Defensor defensor;
private Láser láser;
private Bomba bomba;
```

Cuando un nuevo juego está a punto de empezar, creamos nuevos objetos e iniciamos el temporizador:

```
private void nuevoJuego() {
 defensor = new Defensor();
 extraterrestre = new Extraterrestre();
 temporizador.start();
}
```

Después están los métodos que manejan los eventos:

```
private void temporizador_Tick() {
 if (bomba == null) {
 bomba = new Bomba(extraterrestre.getX(), extraterrestre.getY());
 }
 moverTodo();
 dibujarTodo();
 comprobarChoques();
}

public void mouseClicked(MouseEvent event) {
 int xInicial = defensor.getX();
 int yInicial = defensor.getY();
 if (láser == null) {
 láser = new Láser(xInicial, yInicial);
 }
}

public void mouseMoved(MouseEvent event) {
 defensor.mover(event.getX());
}
```

Luego están los métodos subsidiarios:

```
private void moverTodo() {
 extraterrestre.mover();
 if (bomba != null) {
 bomba.mover();
 }
 if (láser != null) {
 láser.mover();
 }
}

private void dibujarTodo() {
 Graphics papel = panel.getGraphics();
 papel.setColor(Color.white);
 papel.fillRect(0, 0, panel.getWidth(), panel.getHeight());
 defensor.dibujar(panel);
 extraterrestre.dibujar(panel);
 if (láser != null) {
 láser.dibujar(panel);
 }
 if (bomba != null) {
 bomba.dibujar(panel);
 }
}

private void comprobarChoques() {
 if (colisiona(láser, extraterrestre)) {
 terminarJuego("defensor");
 }
 else {
 if (colisiona(bomba, defensor)) {
 terminarJuego("extraterrestre");
 }
 }
 if (bomba != null) {
 if (bomba.getY() > panel.getHeight()) {
 bomba = null;
 }
 }
 if (láser != null) {
 if (láser.getY() < 0) {
 láser = null;
 }
 }
}
```

```

private boolean colisiona(Sprite uno, Sprite dos) {
 if (uno == null || dos == null) {
 return false;
 }
 if (uno.getX() > dos.getX()
 && uno.getY() < (dos.getY() + dos.getAltura())
 && (uno.getX() + uno.getAnchura()) <
 (dos.getX() + dos.getAnchura())
 && (uno.getY() + uno.getAnchura()) > (dos.getY())) {
 return true;
 }
 else {
 return false;
 }
}

```

Y por último:

```

private void terminarJuego(String ganador) {
 láser = null;
 bomba = null;
 temporizador.stop();
 JOptionPane.showMessageDialog(null
 "fin del juego - " + ganador + " gana");
}

```

En este diseño, la clase **Juego** se encarga de una gran parte del trabajo del programa. Por ejemplo, inicia un láser y una bomba. También verifica si el extraterrestre o el defensor chocaron con algo (utilizando el método de propósito general **colisiona**). Éstos son verbos que identificamos en el análisis de la especificación anterior.

## ● En búsqueda de la reutilización

La siguiente etapa del diseño es asegurarnos de no estar reinventando la rueda. Uno de los principales objetivos de la POO es promover la reutilización de los componentes de software. En esta etapa debemos comprobar si:

- Lo que necesitamos podría estar en una de las bibliotecas.
- Tal vez hayamos escrito una clase el mes pasado y sea lo que necesitamos.
- Tal vez podamos generalizar algunas de las clases que hemos diseñado para nuestro programa y obtener una clase más general de la que podamos heredar.

En el programa del invasor del ciberespacio podemos hacer buen uso de los componentes de GUI tales como el panel, disponible en la biblioteca de Java. Otros componentes útiles de la biblioteca son el temporizador y un generador de números aleatorios.

Si encuentra una clase existente que sea similar a lo que necesita, piense en usar la herencia para personalizarla y que haga lo que usted quiere. En el capítulo 10 vimos cómo escribir el código para obtener la herencia. Ahora vamos a examinar una metodología para explorar las relaciones entre las clases mediante el uso de las pruebas “es un” y “tiene un”.

## ● ¿Composición o herencia?

Una vez que identificamos las clases en un programa, el siguiente paso es revisar las relaciones entre ellas. Las clases que conforman un programa colaboran entre sí para obtener el comportamiento requerido, pero se utilizan unas a otras en distintas formas. Hay dos maneras en que las clases se relacionan entre sí:

1. **Composición:** un objeto crea otro objeto a partir de una clase utilizando `new`. Un ejemplo es un marco que crea un botón. Otro ejemplo pueden ser las relaciones entre las clases en el programa del juego que se muestra en la figura 18.4.
2. **Herencia:** una clase hereda de otra. Un ejemplo es una clase que extiende la clase de biblioteca `JFrame` (todos los programas de este libro lo hacen).

La tarea fundamental del diseño es saber diferenciar entre estos dos casos de manera que podamos aplicar o evitar con éxito la herencia. Una manera de comprobar que hemos identificado en forma correcta las relaciones apropiadas entre las clases es utilizar la prueba “es un” o “tiene un”:

- El uso de la frase “es un” en la descripción de un objeto (o clase) significa que probablemente es una relación de herencia.
- El uso de la frase “tiene un” indica que no hay relación de herencia, sino una relación de composición (una frase alternativa que tiene el mismo significado es “consiste en”).

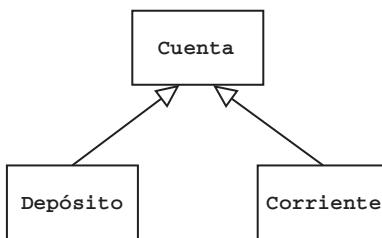
Ahora veamos un ejemplo sobre cómo identificar la herencia. En la especificación de un programa para dar soporte a las transacciones en un banco, descubrimos la siguiente información:

Una cuenta bancaria consiste en el nombre de una persona, su dirección, número de cuenta y saldo actual. Hay dos tipos de cuenta: corriente y de depósito. Los prestatarios tienen que avisar con una semana de anticipación si desean retirar de una cuenta de depósito, pero la cuenta acumula intereses.

Si parafraseamos esta especificación, podríamos decir que “una cuenta corriente es una cuenta bancaria” y “una cuenta de depósito es una cuenta bancaria”. Podemos ver las palabras cruciales “es una”, lo que nos indica que una cuenta bancaria es la superclase tanto de la cuenta de depósito como de la cuenta corriente. Cada una de estas cuentas son subclases de cuenta. Heredan los métodos de la superclase; por ejemplo, un método para obtener la dirección y un método para actualizar el saldo.

En la figura 18.5 se muestra un diagrama de clases útil para describir las relaciones de herencia entre las clases.

Considere también el ejemplo de una ventana: “la ventana tiene un botón y un campo de texto”. Ésta es una relación “tiene un”, la cual es composición y no herencia. La clase que representa a la ventana simplemente declara y crea objetos `JButton` y `JTextField`, y después los utiliza.



**Figura 18.5** Diagrama de clases para las cuentas bancarias.

Ahora regresemos al programa del juego para tratar de encontrar relaciones de herencia. Si podemos encontrar dichas relaciones, podremos simplificar y acortar el programa mediante la reutilización. En el programa del juego, varias de las clases (**Defensor**, **Extraterrestre**, **Láser** y **Bomba**) incorporan los mismos métodos. Estos métodos son **getX**, **getY**, **getAltura** y **getAnchura**, los cuales obtienen la posición y el tamaño de los objetos gráficos. Vamos a eliminar estos ingredientes de cada clase para colocarlos en una superclase. A esta clase la llamaremos **Sprite**, ya que la palabra “sprite” (duendecillo) es un término que se utiliza con frecuencia para describir a un objeto gráfico móvil en la programación de juegos. El diagrama de UML para la clase **Sprite** es:

| clase Sprite           |
|------------------------|
| Variables de instancia |
| x                      |
| y                      |
| altura                 |
| anchura                |
| Métodos                |
| getX                   |
| getY                   |
| getAltura              |
| getAnchura             |

Este diseño se aclara al momento de analizar el código. A continuación le mostramos el código de Java para la clase **Sprite**:

```

public class Sprite {
 protected int x, y, anchura, altura;
 public int getX() {
 return x;
 }
}

```

```

public int getY() {
 return y;
}

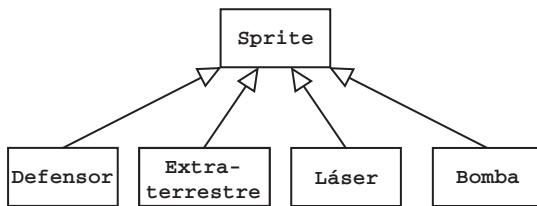
public int getAnchura() {
 return anchura;
}

public int getAltura() {
 return altura;
}

}

```

Las clases **Defensor**, **Extraterrestre**, **Láser** y **Bomba** ahora heredan estos métodos de la clase **Sprite**. Para comprobar la validez de este diseño decimos que “cada una de las clases **Defensor**, **Extraterrestre**, **Láser** y **Bomba** es un **Sprite**”. La figura 18.6 muestra estas relaciones en un diagrama de clases. Recuerde que la flecha apunta de una subclase a una superclase.



**Figura 18.6** Diagrama de clases para los componentes heredados en el juego.

Finalmente hemos podido identificar correctamente las relaciones de herencia entre las clases en el programa del juego.

Ahora vamos a enfocarnos en los detalles del programa. El código para la clase **Extraterrestre** es:

```

import java.awt.*;
import javax.swing.*;

public class Extraterrestre extends Sprite {

 private int tamañoIntervalo;
 private ImageIcon imagenExtraterrestre;

 public Extraterrestre() {
 x = 0;
 y = 25;
 anchura = 20;
 altura = 10;
 tamañoIntervalo = 10;
 imagenExtraterrestre = new ImageIcon("c:\\\\extraterrestre.jpg");
 }

 public void dibujar(JPanel panel) {
 Graphics papel = panel.getGraphics();
 imagenExtraterrestre.paintIcon(panel, papel, x, y);
 }
}

```

```

public void mover() {
 if (x > 200 || x < 0) {
 tamañoIntervalo = -tamañoIntervalo;
 }
 x = x + tamañoIntervalo;
}

```

El código para la clase **Bomba** es:

```

import java.awt.*;
import javax.swing.*;
public class Bomba extends Sprite {
 private int tamañoIntervalo;
 private ImageIcon imagenBomba;
 public Bomba(int xInicial, int yInicial) {
 x = xInicial;
 y = yInicial;
 anchura = 5;
 altura = 10;
 tamañoIntervalo = 10;
 imagenBomba = new ImageIcon("bomba.jpg");
 }
 public void dibujar(JPanel panel) {
 Graphics papel = panel.getGraphics();
 imagenBomba.paintIcon(panel, papel, x, y);
 }
 public void mover() {
 y = y + tamañoIntervalo;
 }
}

```

## PRÁCTICA DE AUTOEVALUACIÓN

### 18.2 Escriba el código para la clase **Láser**.

Para resumir, los dos tipos de relaciones entre clases son:

| Relación entre clases | Prueba                     | El código de Java requiere |
|-----------------------|----------------------------|----------------------------|
| herencia              | "es un"                    | <b>extends</b>             |
| composición           | "tiene un" o "consiste en" | <b>new</b>                 |

## PRÁCTICA DE AUTOEVALUACIÓN

- 18.3 Analice las relaciones entre los siguientes grupos de clases (¿son del tipo “es un” o “tiene un”?):
1. casa, puerta, techo, vivienda
  2. persona, hombre, mujer
  3. automóvil, pistón, caja de cambios, motor
  4. vehículo, automóvil, autobús

## ● Lineamientos para el diseño de clases

El uso de la metodología de diseño que hemos descrito no nos garantiza obtener un diseño perfecto. Siempre es conveniente comparar el diseño de cada clase con los siguientes lineamientos.

### Mantenga los datos privados

Las variables siempre se deben declarar como **private** (o algunas veces como **protected**), pero nunca como **public**. Esto mantiene el ocultamiento de datos, uno de los principios centrales de la POO. Si es necesario acceder a los datos o modificarlos, se debe hacer a través de los métodos que se proporcionan como parte de la clase.

### Inicialice los datos

Aunque Java inicializa de manera automática las variables de instancia (pero no las variables locales) con valores específicos, es una buena práctica inicializarlas de manera explícita, ya sea dentro de la misma declaración de datos o a través de un método constructor.

### Evite clases extensas

Si una clase tiene más de dos páginas de texto, tenemos que considerar dividirla en dos o más clases de menor tamaño. Pero sólo debemos hacer esto si podemos identificar claramente las clases que se puedan formar a partir de la clase más grande. Es contraproducente dividir una clase consistente y estética en clases artificiosas y desagradables.

### Use nombres representativos para las clases y los métodos

Esto hará que sean más fáciles de usar y más atractivos para la reutilización.

### No invente la herencia

En el programa de juegos anterior podríamos crear una superclase llamada **SeMueveHorizontalmente** y hacer a **Defensor** y **Extraterrestre** subclases de ésta. De manera similar, una clase llamada **SeMueveVerticalmente** podría ser la superclase de **Láser** y **Bomba**. Pero si consideramos los requerimientos individuales de estas clases, descubriremos que son bastante distintas y no ganamos nada con ello.

Usar la herencia cuando no es realmente apropiada puede llevarnos a obtener clases artificiosas, que son más complejas y tal vez más extensas de lo necesario.

## Al utilizar la herencia, coloque los elementos compartidos en la superclase

En el ejemplo de la cuenta bancaria que vimos antes, todas las variables y los métodos comunes para todas las cuentas bancarias se deben colocar en la superclase, para que todas las subclases puedan compartirlos sin tener que duplicarlos. Algunos ejemplos de esto son los métodos para actualizar la dirección y para actualizar el saldo.

También vimos en el programa del juego que podemos identificar métodos y variables idénticos en varias de las clases y, por ende, creamos una superclase llamada `Sprite`.

## Use la refactorización

Después de crear un diseño inicial, o cuando hemos realizado cierta parte de la codificación, un estudio del diseño podría revelar que es posible realizar cierta simplificación. Para ello tal vez tengamos que cambiar algunos métodos a otra clase. Tal vez tengamos que crear nuevas clases o amalgamar las clases existentes. A este proceso se le conoce como *refactorización*. Ya hemos cumplido con un lineamiento para la refactorización: colocar los métodos compartidos en la superclase.

Por ejemplo, en el programa del juego existe la necesidad obvia de evaluar varias veces si los objetos chocan. Pero hay varios lugares alternativos en el programa en donde se puede llevar a cabo esta detección de colisiones. Según la implementación que realizamos antes, hay un método llamado `colisiona`, el cual forma parte de la clase `Juego` que lleva a cabo la detección de colisiones. Pero una alternativa sería crear una clase distinta llamada `DetecciónColisiones`, que provea un método estático llamado `colisiona` para llevar a cabo la detección de colisiones. Además, podríamos dispersar el inicio de la detección de colisiones en todo el programa, en las clases `Láser` y `Bomba`.

La refactorización reconoce que a menudo no es posible crear un diseño inicial que sea ideal. En vez de ello, el diseño algunas veces evoluciona hacia una estructura óptima. Para ello tal vez sea necesario cambiar el diseño una vez avanzada la codificación. Por ende, el desarrollo no siempre se lleva a cabo en distintas etapas.

### Resumen

La tarea del diseño OO consiste en identificar los objetos y las clases apropiados. Los pasos en la metodología para el diseño OO que vimos en este capítulo son:

1. Estudiar la especificación y aclararla si es necesario.
2. Obtener los objetos y los métodos a partir de la especificación, de manera que el diseño actúe como un modelo de la aplicación. Los verbos son métodos y los sustantivos son objetos.
3. Generalizar los objetos en clases.
4. Comprobar si es posible reutilizar clases de biblioteca y otras clases existentes a través de la composición y la herencia, según sea apropiado. Los análisis “es un” y “tiene un” nos ayudan a comprobar si es apropiado usar la herencia o la composición.
5. Usar los lineamientos para refinar un diseño.

## Ejercicios

- 18.1** Complete el desarrollo del programa del invasor del ciberespacio.
- 18.2** Mejore el programa del invasor del ciberespacio de manera que el extraterrestre pueda lanzar varias bombas y el defensor pueda disparar varios láseres al mismo tiempo.
- 18.3** Un buen diseño se puede juzgar en cuanto a qué tan bien se acopla a las modificaciones o mejoras. Considere las siguientes mejoras al programa del invasor del ciberespacio. Evalúe los cambios que sean necesarios en el diseño y la codificación:
- Una fila de extraterrestres.
  - Una línea de búnkeres que protejan al defensor de las bombas.
  - Que se muestre el desempeño del jugador.
- 18.4** Un diseño alternativo para el programa del invasor del ciberespacio utiliza una clase llamada **AdministradorPantalla**, la cual:
- Mantiene la información sobre todos los objetos que aparecen en la pantalla.
  - Llama a todos los objetos para que se muestren a sí mismos.
  - Detecta las colisiones entre pares de objetos en la pantalla.
- Rediseñe el programa para que utilice esta clase.

- 18.5 Calculadora** Diseñe las clases para un programa con la siguiente especificación:

El programa debe actuar como una simple calculadora de escritorio (figura 18.7) que opere con números enteros. Debe haber un campo de texto para representar la pantalla. La calculadora debe tener un botón para cada uno de los 10 dígitos, del 0 al 9. Debe contar con un botón para sumar y uno para restar. También debe tener un botón “borrar” para borrar la pantalla y un botón de igual (=) para obtener la respuesta.

Cuando se oprima el botón “borrar”, la pantalla deberá mostrar un 0 y el total (oculto) se deberá establecer en 0.

Cuando se oprima un botón de un dígito, se agregará el dígito correspondiente a la derecha de los dígitos que ya se encuentren en la pantalla (en caso de haber alguno).

Cuando se oprima el botón +, el número en la pantalla se sumará al total (y se restará en caso de que se utilice el botón -).

Cuando se oprima el botón de igual, se mostrará el valor del total.



**Figura 18.7** La calculadora de escritorio.

## Respuestas a las prácticas de autoevaluación

18.1

|                        |
|------------------------|
| Clase Bomba            |
| Variables de instancia |
| x                      |
| y                      |
| altura                 |
| anchura                |
| intervaloY             |
| Métodos                |
| Bomba                  |
| mover                  |
| mostrar                |
| getX                   |
| getY                   |
| getAltura              |
| getAnchura             |

18.2

```
import java.awt.*;
import javax.swing.*;

public class Láser extends Sprite {
 private int tamañoIntervalo;
 private ImageIcon imagenLáser;
 public Láser(int nuevaX, int nuevaY) {
 x = nuevaX;
 y = nuevaY;
 anchura = 5;
 altura = 5;
 tamañoIntervalo = 10;
 imagenLáser = new ImageIcon("c:\\láser.jpg");
 }
 public void dibujar(JPanel panel) {
 Graphics papel = panel.getGraphics();
 imagenLáser.paintIcon(panel, papel, x, y);
 }
 public void mover() {
 y = y - tamañoIntervalo;
 }
}
```

18.3 1. tiene un

2. es un

3. tiene un

4. es un

# CAPÍTULO 19



## Estilo de los programas

En este capítulo le sugerimos lineamientos de estilo para:

- La distribución de un programa.
- Los nombres.
- Las clases.
- Los comentarios.
- Las constantes.
- Los métodos.
- Las instrucciones `if` anidadas.
- Los ciclos anidados.
- Las condiciones complejas.
- La documentación.

### ● Introducción

La programación es una actividad muy creativa y emocionante. A menudo los programadores se absorben en su trabajo y consideran que los programas que producen son sus creaciones muy personales. El estereotipo del programador (hombre o mujer) usa pantalones de mezclilla y una camiseta. Bebe 20 tazas de café al día y permanece despierto toda la noche sólo por la diversión de programar.

Pero los hechos de la vida de los programadores son con frecuencia muy distintos. La mayor parte de la programación se lleva a cabo dentro de organizaciones comerciales. En la mayoría de los programas trabajan varias personas. Muchas organizaciones tienen manuales de estándares que detallan cómo debe ser la apariencia de los programas.

La mayoría de los programas son leídos por distintas personas, y sin duda no sólo por su autor. Los demás pueden ser: la persona que se hace cargo de su trabajo cuando usted recibe un ascenso o se cambia a otro proyecto, las personas que evaluarán su programa y las generaciones de programadores que velarán por él, corrigiendo errores y mejorándolo mucho tiempo después de que usted haya obtenido otro empleo. Por lo tanto, hacer que su programa sea fácil de leer es un ingrediente vital de la programación.

Otro aspecto del buen estilo es la capacidad de reutilización. Un programa que exhibe estilo debe contener clases que se puedan reutilizar más tarde en otro programa, ya sea que lo escriba la misma persona o alguien más.

A menos que usted sea un aficionado, en la práctica es importante saber cómo producir programas que tengan buen estilo.

## Distribución del programa

El programador de Java tiene mucho espacio para decidir cómo distribuir un programa. El formato del lenguaje es libre: se pueden usar líneas y espacios en blanco casi en cualquier parte. Se pueden colocar comentarios en una línea por sí solos o al final de una línea de código. Hay mucho espacio para la creatividad e individualidad.

Sin embargo, y como hemos visto, la mayoría de los programas son leídos por varias personas además del autor original, por lo tanto, es imprescindible que el programa tenga una buena apariencia. A continuación analizaremos un conjunto de lineamientos de estilo para los programas de Java. Siempre hay controversia en cuanto a los lineamientos de este tipo. Sin duda, el lector estará en desacuerdo con algunos de ellos.

### Sangría

La sangría enfatiza la estructura del programa. Hay varios estilos de sangría, de los cuales sólo hemos usado uno en todo el libro. Generalmente las personas también están en desacuerdo en la cantidad de espacios que se deben usar para la sangría; en este libro usamos cuatro.

### Líneas en blanco

Por lo general, se utilizan líneas en blanco para separar visualmente los métodos. También se utilizan a menudo dentro de una clase para separar las declaraciones de las variables de las declaraciones de los métodos, así como para separar un método de otros métodos.

Si hay muchas declaraciones de variables, también se pueden separar distintos bloques de datos mediante líneas en blanco.

## Nombres

En Java, el programador asigna nombres a las variables, las clases y los métodos. Hay mucho espacio para la imaginación, ya que:

- Los nombres pueden constar de letras y dígitos.
- Pueden tener la longitud que usted deseé.

Siempre y cuando el nombre empiece con una letra y se eviten las palabras clave de Java.

Nuestro consejo en cuanto a los nombres es hacerlos lo más representativos que sea posible. De esta forma, quedan descartados los nombres enigmáticos como i, j, x e y, que por lo general indican que el programador tiene cierta experiencia con las matemáticas (pero no mucha imaginación para crear nombres representativos).

La convención para las variables y los métodos es que deben empezar con una letra minúscula y deben usar una letra mayúscula cuando el nombre conste de dos o más palabras juntas. Algunos ejemplos son `miGlobo` y `elMayorSalario`.

La convención para los nombres de las clases es que deben empezar con una letra mayúscula. Algunos ejemplos son `String` y `Globo`. Esto permite al lector diferenciar con facilidad los nombres de las clases de los otros nombres en un programa, sin tener que buscar sus declaraciones.

Por convención, los nombres de los paquetes empiezan con una letra minúscula.

## Clases

Las clases son un elemento fundamental en los programas OO. También son la unidad que facilita la reutilización de los componentes de software (mediante la creación de instancias o la herencia). Por lo tanto, es importante que las clases tengan buen estilo. He aquí algunos lineamientos:

### Modularidad

Un programa de Java se construye como una colección de objetos creados a partir de clases. El buen diseño de las clases nos ayuda a asegurar que el programa sea claro y comprensible, debido a que consta de varios módulos. En el capítulo sobre diseño orientado a objetos (DOO) vimos una metodología para el buen diseño.

### Complejidad

El DOO trata de diseñar clases que correspondan con las clases del problema que buscamos resolver. Por lo general, estas clases están presentes en la especificación para el programa. Por lo tanto, en un buen diseño debemos ser capaces de reconocer las clases como un modelo de la especificación. Como producto derivado, el diseño debe reflejar la complejidad del problema y nada más.

### Ocultamiento de datos

La idea del DOO es ocultar o encapsular los datos, de manera que cualquier interacción entre las clases se realice a través de los métodos en vez de acceder directamente a los datos. Un buen diseño de una clase tiene un mínimo de variables públicas.

### Tamaño de la clase

Si, por decir, una clase ocupa más de dos páginas, tal vez sea demasiado grande y compleja. Considere dividirla (cuidadosamente) en dos o más clases, de tal forma que pueda crear nuevas clases viables. Sin embargo, es peligroso dividir una clase coherente en varias clases burdas e incoherentes. Al proceso de reestructurar una clase se le denomina *refactorización*.

## Orden de los campos

Los campos son las variables y los métodos declarados dentro de una clase. ¿En qué orden deben aparecer? Tenemos que considerar tanto a los campos **public** como **private**. La convención que se adopta en las bibliotecas de Java y en todos los libros autorizados sobre Java es escribirlos en el siguiente orden:

1. Variables de instancia (**public** y **private**).
2. Métodos **public**.
3. Métodos **private**.

## Comentarios

Hay dos formas principales de colocar comentarios en programas de Java:

```
// éste es un comentario al final de la línea

/* éste es un comentario
que abarca
varias líneas */
```

Siempre hay una gran controversia en cuanto a los comentarios en los programas. Algunas personas argumentan que “entre más sean, mejor”. Sin embargo, algunas veces podemos encontrarnos con código como el siguiente:

```
// muestra el mensaje "hola"
campoTexto.setText("hola");
```

en donde el comentario simplemente repite lo que el código implica, y por lo tanto es superfluo.

Algunas veces el código se ve invadido por comentarios sofocantes que ayudan muy poco a comprender mejor su significado. Es como un árbol de Navidad que se ve invadido de guirnaldas, adornos y luces; no podemos ver el árbol por tantas decoraciones. También tenemos otro problema: algunos estudios han demostrado que cuando hay muchos comentarios, el lector lee los comentarios e ignora el código. Por lo tanto, si el código es incorrecto, así permanecerá.

Algunas personas argumentan que los comentarios son necesarios cuando el código es complejo o difícil de entender en cierta forma. Esto parece razonable hasta que nos preguntamos por qué el código necesita ser complejo en primer lugar. Tal vez algunas veces el código se pueda simplificar de manera que sea fácil de comprender sin comentarios. A continuación le mostraremos ejemplos de dichas situaciones.

Algunos programadores prefieren colocar un comentario al inicio de cada clase y, a veces, de un método, para describir el propósito general de la clase (o del método). Las clases de la biblioteca de Java están documentadas de esta forma. Desde luego, los nombres de las clases y los métodos deberían procurar describir lo que hacen, por lo que un comentario podría ser redundante.

Nosotros le recomendamos utilizar los comentarios con juicio y medida.

## Javadoc

Hay una herramienta de software conocida como Javadoc (el Generador de Documentación de Java) que explora el texto de un programa y produce un documento de HTML (Lenguaje de Marcado de Hipertexto). El programa muestra la jerarquía de herencia de las clases, junto con las variables y los métodos públicos que contiene cada clase. También detecta e incorpora el siguiente estilo de comentarios (cuando se asocian con una clase, un método o una variable) y los utiliza en el informe. Podemos ver este informe en un navegador Web.

```
/** este estilo de comentario
es para usarse junto con Javadoc para
producir documentación automática
*/
```

## Constantes

Muchos programas tienen valores que no cambian mientras el programa está en ejecución (y de todas formas no cambian con mucha frecuencia). Algunos ejemplos son los impuestos de ventas, la edad para votar, el monto límite para pagar impuestos y las constantes matemáticas. Java nos da la facilidad de declarar elementos de datos como constantes y asignarles un valor. Entonces, si tomamos en cuenta los ejemplos antes mencionados, podemos escribir:

```
final double tasaImpuestos = 17.5;
final int edadVotar = 18;
final int límiteImpuestos = 5000;
final double pi = 3.142;
```

Las variables como éstas con valores constantes sólo se pueden declarar en la parte superior de una clase y no como variables locales dentro de un método.

Las cadenas de texto y los arreglos también pueden recibir valores constantes:

```
final string nuestroPlaneta = "Tierra";
final int [] precios = {12, 18, 24};
```

Un beneficio de usar valores **final** es que el compilador detectará cualquier intento (sin duda erróneo) de cambiar el valor de una constante. Por ejemplo, dada la anterior declaración de **edadVotar**, la siguiente línea de código:

```
edadVotar = 17;
```

provocará un mensaje de error del compilador.

Otro beneficio, más relevante, es que un programa, en vez de estar invadido de números nada significativos, puede contener variables (que sean constantes) con nombres claros y significativos. Esto mejora la claridad del programa, con todos sus beneficios consecuentes.

Por ejemplo, suponga que necesitamos modificar un programa de impuestos para reflejar un cambio en las regulaciones. Nuestra tarea será una pesadilla si todos los límites y las tasas de impuestos están integrados en el programa como números que aparecen cada vez que se requieren en diversas

partes del mismo. Suponga que el monto límite de impuestos anterior es de **5000**. Podríamos usar un editor de texto para buscar todas las ocurrencias de **5000**. El obediente editor nos indicará en dónde se encuentran todas las ocurrencias, pero no podemos estar seguros de que este número tenga el mismo significado en todas partes. ¿Qué tal si aparece el número **4999** en el programa? ¿Será el umbral de impuestos – 1? ¿O tiene algún otro significado que no esté relacionado? Desde luego, la respuesta es utilizar constantes con nombres descriptivos y asegurarnos de tener cuidado al diferenciar los distintos elementos de datos.

Otro uso común de los valores **final** es para especificar el tamaño de los arreglos utilizados en un programa, como en el siguiente ejemplo:

```
final int tamañoArreglo = 10;
```

y, por consiguiente:

```
int[] miArreglo = new int[tamañoArreglo];
```

La convención es usar letras mayúsculas para los nombres de las constantes que se proveen en las bibliotecas. Por ejemplo, **PI**, **E**, **HORIZONTAL**. El uso de mayúsculas diferencia estos valores de los demás en el programa y hace evidente que en definitiva son constantes. La mayoría de los programadores utilizan esta misma convención para los nombres de sus propias constantes.

## Métodos

### Nombres de los métodos

Ya hemos recalco la importancia de los nombres representativos. Los nombres de los métodos son casi siempre verbos. Cuando un método tiene la simple función de obtener el valor de un elemento de datos (por decir, de **salario**), es convencional llamarlo **getSalario**. De manera similar, si se va a proveer un método para modificar el valor de esta misma variable, entonces el nombre por convención es **setSalario**.

### Tamaño de los métodos

Es posible enfascarnos en discusiones extensas y divertidas en cuanto a la longitud apropiada de los métodos.

Una de las opiniones es que un método no debe ser más grande que el tamaño de la pantalla (alrededor de 40 líneas de texto). De esa forma no tenemos que desplazarnos para estudiarlo completo. Podemos estudiar el método detalladamente en su totalidad. No es tan grande como para perder la pista de algunas partes del mismo.

Cualquier método mayor de media página se considera un serio candidato a ser reestructurado en varios métodos más pequeños. Sin embargo, esto depende de lo que el método haga; tal vez realice una sola tarea cohesiva, y si tratamos de dividirlo cabe la posibilidad de que surjan complicaciones en cuanto a los parámetros y el alcance. No aplique a ciegas ninguna recomendación relacionada con la longitud de un componente.

Si ve los programas en la pantalla, puede ser útil restringir la longitud de los métodos a lo que se pueda ver en una pantalla.

## ● Instrucciones **if** anidadadas

Anidar es el término que se da a la situación dentro de un programa cuando una instrucción se encuentra dentro de otra; por ejemplo, un ciclo **while** dentro de un ciclo **for** o una instrucción **if** dentro de otra instrucción **if**. Algunas veces un programa anidado es simple y claro. Pero en general, un alto nivel de anidamiento se considera un mal estilo y es mejor evitarlo. Siempre es posible evitar el anidamiento si reestructuramos el programa.

Ahora veamos las instrucciones **if** anidadadas. Considere el problema de encontrar el mayor de tres números:

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

if (a > b) {
 if (a > c) {
 mayor = a;
 }
 else {
 mayor = c;
 }
}
else {
 if (b > c) {
 mayor = b;
 }
 else {
 mayor = c;
 }
}
```

Sin duda, este programa se ve complicado, y tal vez a algunas personas se les dificulte un poco comprenderlo. La gente no siempre está convencida de que funciona en forma correcta. Por lo tanto, con base en la evidencia, es difícil de leer y comprender. Sin duda, la complejidad se debe al anidamiento de las instrucciones **if**.

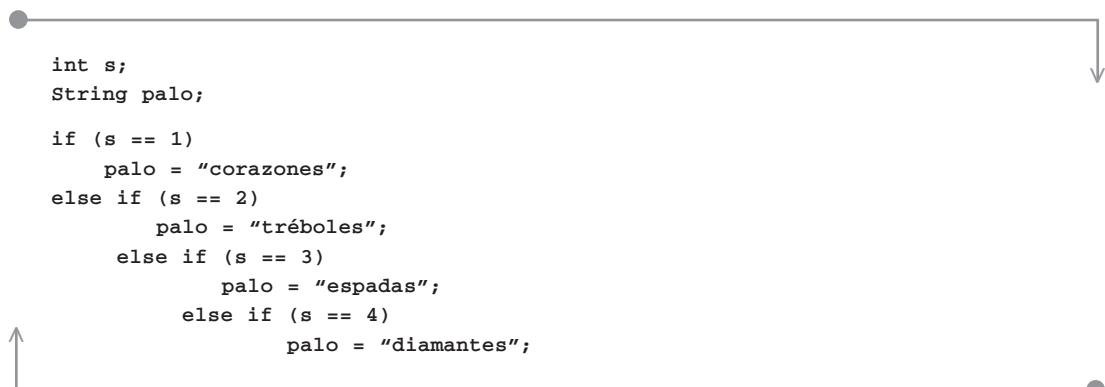
He aquí un fragmento de programa opcional que evita el anidamiento:

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

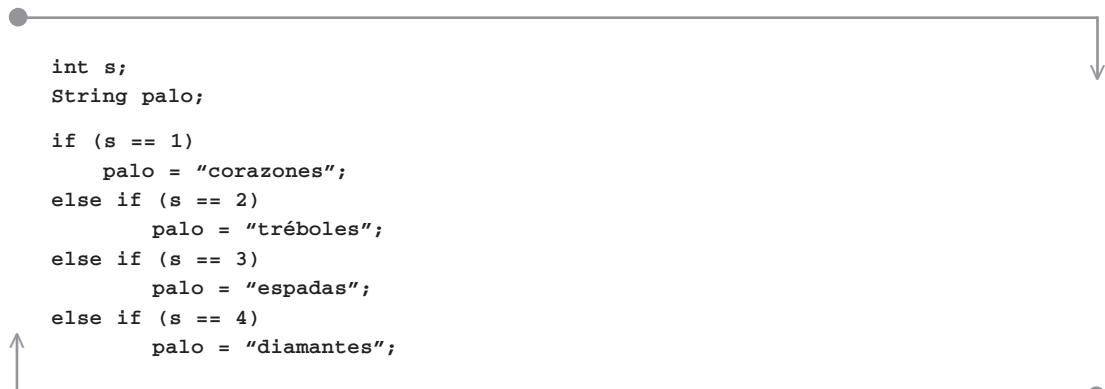
if (a > b && a > c) {
 mayor = a;
}
if (b > a && b > c) {
 mayor = b;
}
if (c > a && c > b) {
 mayor = c;
}
```

Este programa puede ser más legible para algunas personas. El problema con esta solución sin anidamiento es que las tres instrucciones `if` se ejecutan **siempre**, mientras que en el primer programa sólo se realizan **dos** evaluaciones. Por ende, el segundo programa se ejecutará con un poco más de lentitud. Esto es verdad en general; los programas con instrucciones `if` anidadas se ejecutan más rápido.

He aquí otro ejemplo de anidamiento. En un programa para participar en un juego de cartas, el palo de la carta está codificado como número entero (del 1 al 4). En esta parte del programa, el entero se convierte al nombre de cadena de texto del palo. Este código utiliza varias instrucciones `if` anidadas. Recuerde que cuando sólo hay una instrucción dentro del `if`, no son necesarias las llaves. Así, podemos escribir el código en forma más concisa mediante el siguiente estilo:



en donde hay un `if` dentro de un `if`, dentro de un `if`, etc. Esta pieza de programa utiliza una sangría consistente, pero tal vez no sea fácil de entender. Por lo tanto, algunas personas recomiendan distribuir este código de la siguiente manera:



lo cual es más compacto, pero no muestra el anidamiento con tanta claridad.

Una alternativa es escribir el código sin anidamiento, como se muestra a continuación:

```
int s;
String palo;

if (s == 1) {
 palo = "corazones";
}
if (s == 2) {
 palo = "tréboles";
}
if (s == 3) {
 palo = "espadas";
}
if (s == 4) {
 palo = "diamantes";
}
```

lo cual, sin duda, es mucho más claro. De nuevo, la desventaja es que el programa más claro es más lento, ya que siempre se ejecutan todas las instrucciones **if**.

Sin duda, usted, el lector, habrá visto que puede haber una solución para este dilema. Podríamos recodificar esta pieza de programa mediante la instrucción **switch**, como en el siguiente ejemplo:

```
int s;
String palo;

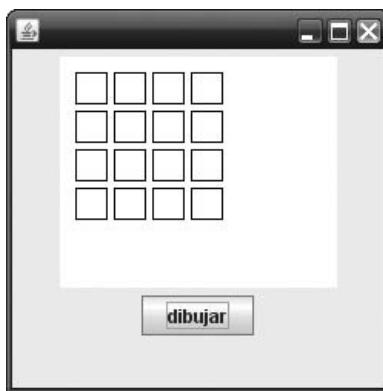
switch(s) {
 case 1 : palo = "corazones"; break;
 case 2 : palo = "tréboles"; break;
 case 3 : palo = "espadas"; break;
 case 4 : palo = "diamantes"; break;
}
```

lo cual es más claro y rápido. Sin embargo, el problema es que la instrucción **switch** tiene limitaciones; sólo podemos utilizarla con un valor entero o tipo char. Así que, por ejemplo, no se puede utilizar en el programa anterior para buscar el mayor de tres números. El uso de **switch** no es una solución general para el problema de las instrucciones **if** anidadas.

La conclusión es ésta: si evita las instrucciones **if** anidadas tal vez sufra un castigo en el rendimiento. En la práctica, la reducción en el rendimiento sólo es considerable si la prueba se lleva a cabo dentro de un ciclo que se repite muchas veces en un programa con requerimientos de tiempo muy estrictos.

Por último, basta decir que las instrucciones **if** anidadadas no son siempre malas e incluso hay ocasiones en que el anidamiento describe con simpleza y claridad lo que se debe hacer.

## Ciclos anidados



**Figura 19.1** Ciclos anidados.

Ahora veamos los ciclos anidados. Suponga que debemos escribir un programa para mostrar un patrón en la pantalla como el de la figura 19.1, un gráfico simple de un bloque de apartamentos. El programa podría ser algo así:

```
private void dibujarApartamentos(int pisos, int apartamentos) {
 int coordX, coordY;
 coordY = 10;
 for (int piso = 0; piso <= pisos; piso++) {
 coordX = 10;
 for (int contador = 0; contador <= apartamentos; contador++) {
 papel.drawRect(coordX, coordY, 20, 20);
 coordX = coordX + 25;
 }
 coordY = coordY + 25;
 }
}
```

En este programa, un ciclo está anidado dentro de otro. No es una pieza compleja de código en sí, pero podemos simplificarla si utilizamos otro método:

```
private void dibujarApartamentos(int pisos, int apartamentos) {
 int coordY = 10;
 for (int contador = 0; contador < pisos; contador++) {
 dibujarPiso(coordY, apartamentos);
 coordY = coordY + 25;
 }
}
```

```
private void dibujarPiso(int coordY, int apartamentos) {
 int coordX = 10;
 for (int contador = 0; contador < apartamentos; contador++) {
 papel.drawRect(coordX, coordY, 20, 20);
 coordX = coordX + 25;
 }
}
```

Gracias al método adicional pudimos eliminar el anidamiento. También expresamos de manera explícita en el código el hecho de que el bloque de apartamentos consiste en varios pisos. Aclaramos el requerimiento de que hay un cambio en la coordenada *y* para cada piso del bloque. Siempre es posible eliminar ciclos anidados de esta forma y algunas veces esto ayuda a simplificar el programa.

Algunos estudios de investigación han mostrado que a los humanos se nos dificulta comprender programas que utilicen anidamiento. Un investigador resumió este hecho al decir: “El anidamiento es para las aves”. Pero no **siempre** es malo. Por ejemplo, considere el código para inicializar una matriz:

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

for (int fila = 0; fila < 10; fila++) {
 for (int col = 0; col < 10; col++) {
 tabla[fila][col] = 0;
 }
}
```

Este ejemplo es claro, aunque use anidamiento.

## ● Condiciones complejas

La complejidad en una instrucción **if**, **for** o **while** puede surgir cuando la condición a evaluar implica uno o más operadores **and** y **or**. Una condición compleja puede hacer que un programa sea muy difícil de entender, depurar y corregir. Como ejemplo analizaremos un programa que busca el número deseado en un arreglo de números:

```
final int longitud = 100;
int[] tabla = new int[longitud];

int buscado;
int índice;

índice = 0;
while (índice < longitud && tabla[índice] != buscado) {
 índice++;
}
```

```

if (índice == longitud) {
 estado.setText("no se encontró");
}
else {
 estado.setText("se encontró");
}

```

El problema con este programa es que la condición en el ciclo `while` es compleja. Incluso para un programador experimentado puede ser difícil comprobar lo que se escribió y convencerse a sí mismo de que es correcto. Hay una alternativa: utilizaremos una bandera. Es simplemente una variable entera, pero su valor en cualquier momento registra el estado de la búsqueda. Hay tres posibles estados en los que se puede encontrar la búsqueda:

- El programa está aún buscando; no se ha encontrado el elemento. Éste es también el estado inicial de la búsqueda. La bandera tiene el valor de 0.
- El elemento se encontró. El valor de la bandera es 1.
- La búsqueda se completó sin encontrar el elemento. El valor de la bandera es 2.

Si utilizamos esta bandera, a la que llamaremos `estado`, el programa queda así:

```

final int longitud = 100;
int[] tabla = new int[longitud];

int buscado;
int estado;
final int sigueBuscando = 0;
final int encontró = 1;
final int noEncontró = 2;

estado = sigueBuscando;
for (int índice = 0; estado == sigueBuscando; índice++) {
 if (buscado == tabla[índice]) {
 estado = encontró;
 }
 else {
 if (índice == longitud - 1) {
 estado = noEncontró;
 }
 }
}

if (estado == noEncontró) {
 estado.setText("no se encontró");
}
else {
 estado.setText("se encontró");
}

```

Lo que logramos aquí es desenmarañar las diversas evaluaciones. La condición en el ciclo `for` es más clara y sencilla. Las demás evaluaciones son separadas y simples. Como se puede apreciar, el programa en general es más simple.

La moraleja es que a menudo es posible escribir una pieza de programa en distintas formas. Algunas soluciones son más simples y claras que otras. En ocasiones es posible evitar la complejidad en una condición reestructurando el fragmento del programa con una bandera.

## ● Documentación

La documentación es la pesadilla del programador... ¡hasta que se le pide que explique el funcionamiento del programa de otra persona! Por lo general, las organizaciones comerciales procuran alentar a los programadores para que documenten bien sus programas. Les cuentan la antigua y tal vez imaginaria historia sobre el programador que tenía el 95% completado de un programa, que no hizo ninguna documentación y más tarde lo atropelló un autobús. Supuestamente los colegas que se hicieron cargo de su trabajo tuvieron muchas dificultades para continuar trabajando en el programa.

Por lo común, la documentación de un programa consiste en los siguientes ingredientes:

- La especificación del programa.
- El código fuente, incluyendo los comentarios apropiados.
- Información de diseño, por ejemplo, los diagramas de clases.
- El calendario de pruebas.
- Los resultados de las pruebas.
- El historial de modificaciones.
- El manual del usuario (si es necesario).

Si alguna vez le piden hacerse cargo del programa de alguien más, es lo que necesitará... ¡Mas no espere obtenerlo!

En general, los programadores encuentran que la creación de documentación es una tarea aburrida y tienden a escatimar. Es común que la dejen hasta el final del proyecto, cuando haya poco tiempo disponible. No es de sorprender que la mayor parte de las veces sea muy deficiente o no se lleve a cabo.

La única forma de aliviar el dolor es realizar la documentación a medida que vamos avanzando, mezclándola con las tareas más interesantes de la programación. En la sección sobre comentarios de este capítulo le sugerimos cómo documentar el código fuente de los programas.

## ● Consistencia

Aunque las opiniones de las personas sobre el estilo de programación difieren la mayor parte del tiempo, algo en lo que siempre están de acuerdo es que un estilo se debe aplicar de manera consistente en todo un programa. Si el estilo es inconsistente, el programa será difícil de leer (por no decir que molesto). También surge la preocupación de que al programador original en realidad no le importaba el programa y de que haya algo mal en él. En todo el libro hemos utilizado un estilo consistente para la distribución de los programas. Es el estilo que se utiliza en las bibliotecas de Java y en los libros autorizados sobre Java, escritos por James Gosling (el principal diseñador de Java) y sus colegas.

## Errores comunes de programación

No invierta horas y horas en embellecer su programa para después descubrir que hay una útil herramienta disponible para ello. La mayoría de los entornos de desarrollo interactivos poseen un programa embellecedor que imprime su código en forma bonita y aplica sangría para mantenerlo ordenado. Verifique qué opciones hay disponibles dentro de su entorno de programación **antes** de empezar a codificar. También averigüe si hay estándares que utilice su empresa antes de empezar a codificar. Tal vez tenga que seguirlos. Si desea apegarse a un plan para elaborar la distribución del programa, siempre es mejor hacerlo desde el principio en vez de apresurarse a escribir el código del programa y cambiarlo después.

## Resumen

- El estilo de un programa es importante, ya que mejora su legibilidad para los procesos de depuración y mantenimiento.
- Algunos lineamientos para una buena distribución de los programas son: nombres apropiados, sangría, líneas en blanco y comentarios.
- La descripción `final` de Java convierte a un elemento de datos en una constante.
- Los métodos no deben ser demasiado extensos.
- Las instrucciones `if`, los ciclos y las condiciones complejas anidadas se deben utilizar con medida.
- Una buena documentación es algo que siempre vale la pena.

## Ejercicios

- 19.1** Analice todos los programas que pueda (incluyendo los suyos) y revise sus estilos. ¿Son buenos o malos? ¿Por qué?
- 19.2** Discuta la cuestión de los lineamientos con sus colegas o amigos. ¿Es importante el estilo? De ser así, ¿qué es lo que constituye un buen estilo?
- 19.3** Investigue qué programas o qué opciones automáticas existen dentro de su entorno de desarrollo para mejorar la apariencia de sus códigos.
- 19.4** Desarrolle un conjunto de lineamientos de estilo para los programas de Java.
- 19.5** (Opcional) Use sus lineamientos de estilo para siempre.

# CAPÍTULO **20**



## El proceso de prueba

En este capítulo conoceremos:

- Por qué no es viable realizar una prueba exhaustiva.
- Cómo llevar a cabo una prueba funcional.
- Cómo realizar una prueba estructural.
- Cómo llevar a cabo recorridos.
- Cómo realizar pruebas paso a paso por instrucciones.
- Cómo se lleva a cabo el desarrollo incremental.

### ● Introducción

Los programas son complejos y es difícil hacer que funcionen correctamente. La prueba de un programa consiste en el conjunto de técnicas que se utilizan para tratar de verificar que un programa funcione en forma correcta. Dicho de otra manera, el proceso de prueba trata de revelar la existencia de errores. Una vez que detectamos un error, necesitamos localizarlo por medio de la depuración (vea el capítulo 21) y después corregirlo. Como veremos más adelante, las técnicas de prueba no pueden garantizar que se expongan todos los errores de un programa y, por lo tanto, la mayoría de los programas extensos tienen errores ocultos. Sin embargo, el proceso de prueba es muy importante debido a que por lo general consume hasta la mitad del tiempo total invertido en el desarrollo de un programa. En algunas organizaciones, el proceso de prueba se considera algo tan importante que hay equipos de programadores (quienes escriben programas) y equipos separados de evaluadores (quienes prueban los programas). Es común que haya la misma cantidad de evaluadores que de programadores.

Como se requiere mucho tiempo y esfuerzo para probar y depurar los programas por completo, a menudo hay que tomar una difícil decisión: si debemos continuar las pruebas o entregar el programa en su estado actual a los clientes.

En los círculos académicos, a la tarea de tratar de asegurar que un programa haga lo esperado se le conoce como *verificación*. El objetivo es *verificar* que un programa cumpla con su especificación.

En este capítulo veremos cómo llevar a cabo el proceso de prueba en forma sistemática, revisaremos distintas metodologías para la verificación y veremos cuáles son sus deficiencias.

Las técnicas que revisaremos son:

- Caja negra o prueba funcional.
- Caja blanca o prueba estructural.
- Revisiones o recorridos.
- Paso a paso por instrucciones con un depurador.

Por lo general, un pequeño programa que sólo contiene una clase se puede probar de una sola vez. Un programa más grande que contenga dos o más clases puede llegar a tener una complejidad tal que debe probarse por piezas. En Java, el tamaño natural de estas piezas es la clase, por lo cual es conveniente probar un programa clase por clase. A esto se le conoce como *prueba unitaria*. A la tarea de reunir el programa completo para evaluarlo se le conoce como *integración* o *prueba de sistema*.

También veremos cómo desarrollar un programa pieza por pieza, en vez de hacerlo como un programa completo.

## ● Especificaciones de los programas

El punto inicial para cualquier prueba es la especificación. El tiempo que dedicamos a estudiar y aclarar la especificación nunca es un desperdicio. Tal vez sea necesario ir con el cliente o con el futuro usuario del programa. Por ejemplo, veamos la siguiente especificación:

Escriba un programa que reciba como entrada una serie de números mediante un cuadro de texto. El programa debe calcular y mostrar la suma de los números.

Al leer la especificación por primera vez, nos puede parecer simple y clara. Pero, aun cuando es tan corta, contiene algunos obstáculos:

- ¿Son los números enteros o de punto flotante?
- ¿Cuáles son el rango y la precisión permisibles de los números?
- ¿Se deben incluir números negativos en la suma?

El programador debe aclarar estas preguntas antes de empezar a trabajar. Sin duda, es parte del trabajo de la programación estudiar la especificación, descubrir si hay omisiones o confusiones y llegar a una especificación clara. Después de todo, no tiene caso escribir un programa brillante si no hace lo que el cliente esperaba.

Ahora veamos una versión más clara de la especificación.

Escriba un programa que reciba como entrada una serie de números enteros mediante un cuadro de texto. Los enteros deben estar en el rango de 0 a 10000. El programa debe calcular y mostrar la suma de los números.

Podemos ver que esta especificación es más precisa; por ejemplo, estipula el rango permisible de los valores de entrada.

## PRÁCTICA DE AUTOEVALUACIÓN

**20.1** ¿Puede ver alguna otra deficiencia en la especificación anterior que debamos aclarar?

### ● Prueba exhaustiva

Una metodología para el proceso de prueba sería evaluar un programa con todos los valores de datos posibles como entrada. Pero consideremos incluso al más simple de los programas que reciba como entrada un par de números enteros y muestre su producto. La prueba exhaustiva implicaría seleccionar todos los valores posibles para el primer número y todos los valores posibles para el segundo. Y después de eso utilizamos todas las posibles combinaciones de números. En Java, un número `int` tiene un enorme rango de valores posibles. A fin de cuentas, la cantidad de posibles combinaciones de los números es enorme. Tendríamos que introducir mediante el teclado todos los distintos valores y ejecutar el programa. El tiempo humano requerido para ensamblar los datos de prueba sería de varios años. Incluso el tiempo requerido por la computadora sería de días; y eso que son rápidas. Por último, el proceso de comprobar que la computadora haya obtenido las respuestas correctas volvería loco de aburrimiento a cualquiera.

Debido a todo lo anterior, no es factible realizar una prueba exhaustiva —ni siquiera para un programa pequeño y simple. Es importante reconocer que es imposible realizar un proceso completo de prueba para todos los programas, excepto para los más pequeños. Por lo tanto, debemos adoptar alguna otra metodología.

### ● Prueba de la caja negra (funcional)

Ahora que sabemos que no es viable realizar una prueba exhaustiva, la metodología de la caja negra para probar programas consiste en idear datos de muestra que representen a todos los datos posibles. Después ejecutamos el programa, introducimos los datos y vemos qué ocurre. A este tipo de prueba se le denomina *prueba de la caja negra* debido a que no necesitamos conocer el funcionamiento del programa para realizar las pruebas; sólo nos concentramos en las entradas y salidas. Consideramos que el programa está completamente encerrado dentro de una caja negra y no lo podemos ver. A la prueba de la caja negra también se le conoce como prueba funcional, ya que sólo hay que conocer la función que desempeña el programa (y no cómo trabaja).

Lo ideal es anotar los datos de prueba y los resultados esperados antes de realizar la prueba. A esto se le conoce como casos de prueba, especificación o calendario de prueba. Después ejecutamos el programa, introducimos los datos y examinamos los resultados para ver si hay discrepancias entre el resultado esperado y el resultado real. Los datos de prueba también deben verificar si el programa maneja las excepciones de acuerdo con su especificación.



**Figura 20.1** El programa verificador de votantes.

Consideremos un programa que decide si una persona puede votar, dependiendo de su edad (figura 20.1). La edad mínima para votar es de 18.

Sabemos que no podemos probar este programa con todos los valores posibles, sino que necesitamos algunos valores representativos. La metodología de idear datos para la prueba de la caja negra es utilizar un *particionamiento equivalente*. Esto significa analizar la naturaleza de los datos de entrada para identificar características comunes. A dicha característica común se le denomina *partición*. En el programa para votar, reconocemos que los datos de entrada se pueden ubicar en dos particiones:

- Los números menores de 18.
- Los números mayores o iguales a 18.

Podemos hacer un diagrama de esto como se muestra a continuación:



Hay dos particiones, una incluye el rango de edades de 0 a 17 y la otra los números del 18 al infinito. Después tenemos que asegurarnos de poder afirmar que todos los números dentro de una partición son equivalentes a cualquier otro para los fines de prueba de este programa (por ello se le llama particionamiento equivalente). Ahora debemos argumentar que el número 12 es equivalente a cualquier otro en la primera partición y que el número 21 es equivalente a cualquier número en la segunda partición. Para ello desarrollamos dos pruebas:

| Número de prueba | Datos | Resultado      |
|------------------|-------|----------------|
| 1                | 12    | no puede votar |
| 2                | 21    | puede votar    |

Hemos llegado a la conclusión de que necesitamos dos conjuntos de datos de prueba para evaluar este programa. Estos dos conjuntos, aunados a una declaración de los resultados esperados, constituyen una especificación de prueba. Ejecutamos el programa con los dos conjuntos de datos y observamos cualquier discrepancia entre los resultados esperados y los reales.

Por desgracia, podemos ver que estas pruebas no han tomado en cuenta la importante distinción entre alguien de 17 años y alguien de 18. Cualquiera que haya escrito un programa sabe que se pueden cometer errores al usar instrucciones `if`, por lo que es recomendable investigar esta región

específica de los datos. Esto es lo mismo que reconocer que vale la pena incluir en la prueba los valores de los datos en los extremos de las particiones. Por ende, creamos dos pruebas adicionales:

| Número de prueba | Datos | Resultado      |
|------------------|-------|----------------|
| 3                | 17    | no puede votar |
| 4                | 18    | puede votar    |

En resumen, las reglas para seleccionar los datos para la prueba de la caja negra mediante partición equivalente son:

1. Particionar los valores de los datos de entrada.
2. Seleccionar datos representativos de cada partición (datos equivalentes).
3. Seleccionar los datos en los límites de las particiones.

En el programa anterior hay una sola entrada; hay cuatro valores de datos y, por lo tanto, cuatro pruebas. Sin embargo, la mayoría de los programas procesan varias entradas. Suponga que deseamos probar un programa que muestra el mayor de dos números, cada uno en el rango de 0 a 10000, y los números se introducen en un par de cuadros de texto. Si los valores son iguales, el programa muestra cualquiera de los dos.

Cada entrada está dentro de una partición que contiene los valores del 0 al 10000. Elegimos los valores en cada extremo de las particiones, junto con valores de muestra en algún punto intermedio:

|                 |   |     |       |
|-----------------|---|-----|-------|
| Primer número:  | 0 | 54  | 10000 |
| Segundo número: | 0 | 142 | 10000 |

Una vez seleccionados los valores representativos, necesitamos considerar qué combinaciones de valores debemos usar. La prueba exhaustiva implica utilizar cada posible combinación de todos los posibles valores de datos; desde luego, esto es imposible. En vez de ello utilizamos todas las combinaciones de los valores representativos. Entonces, las pruebas son:

| Número de prueba | 1er. número | 2do. número | Resultado |
|------------------|-------------|-------------|-----------|
| 1                | 0           | 0           | 0         |
| 2                | 0           | 142         | 142       |
| 3                | 0           | 10000       | 10000     |
| 4                | 54          | 0           | 54        |
| 5                | 54          | 142         | 142       |
| 6                | 54          | 10000       | 10000     |
| 7                | 10000       | 0           | 10000     |
| 8                | 10000       | 142         | 10000     |
| 9                | 10000       | 10000       | 10000     |

De esta forma, el paso adicional en la prueba es utilizar todas las combinaciones de los valores de datos representativos (limitados).

## PRÁCTICA DE AUTOEVALUACIÓN

- 20.2** En un programa para jugar ajedrez, el jugador especifica el destino de un movimiento como un par de índices: los números de fila y de columna. El programa comprueba que el cuadro de destino sea válido; es decir, que no esté fuera del tablero. Desarrolle los datos de una prueba de caja negra para comprobar que esta parte del programa funcione correctamente.

### ● Prueba de la caja blanca (estructural)

En la prueba de la caja blanca debemos saber cómo funciona el programa (su estructura) como base para desarrollar los datos de prueba. En este tipo de prueba cada instrucción del programa se ejecuta en cierto momento durante el proceso. Esto equivale a asegurar que se ejecutarán todas las rutas (todas las secuencias de instrucciones) del programa en algún momento durante el periodo de prueba. Se incluyen las rutas nulas, por lo que una instrucción `if` sin la instrucción `else` equivalente tiene dos rutas y cada ciclo tiene dos rutas. El proceso de prueba también debe incluir el manejo de excepciones que el programa lleve a cabo.

He aquí el código para el programa verificador de votantes que utilizamos como ejemplo práctico:

```
public void actionPerformed(ActionEvent event) {
 int edad;
 edad = Integer.parseInt(campoTexto.getText());
 if (edad >= 18) {
 resultado.setText("puede votar");
 }
 else {
 resultado.setText("no puede votar");
 }
}
```

En este programa hay dos rutas (debido a que la instrucción `if` tiene dos ramificaciones) y, por lo tanto, bastará con dos conjuntos de datos para asegurar que se ejecuten todas las instrucciones en un momento dado durante el periodo de prueba:

| Número de prueba | Datos | Resultado esperado |
|------------------|-------|--------------------|
| 1                | 12    | no puede votar     |
| 2                | 21    | puede votar        |

Si somos cuidadosos, debemos tener en cuenta que a menudo los errores en la programación se cometan dentro de las condiciones de las instrucciones `if` y `while`. Por ende, debemos agregar dos pruebas para asegurar que la instrucción `if` funcione correctamente:

| Número de prueba | Datos | Resultado esperado |
|------------------|-------|--------------------|
| 3                | 17    | no puede votar     |
| 4                | 18    | puede votar        |

Entonces necesitamos cuatro conjuntos de datos para probar este programa con el método de la caja blanca. En este caso son los mismos datos que desarrollamos para la prueba de la caja negra, pero el razonamiento que condujo a los dos conjuntos de datos es distinto. Si el programa se hubiera escrito en forma distinta, los datos de la prueba de la caja blanca hubieran sido diferentes. Por ejemplo, suponga que el programa utiliza un arreglo llamado `tabla` con un elemento para cada edad que especifica si alguien de esa edad puede votar o no. Entonces el programa sólo constará de la siguiente instrucción para ver si el candidato puede votar:

```
resultado.setText(tabla[edad]);
```

y los datos de prueba de la caja blanca serán distintos.

## PRÁCTICA DE AUTOEVALUACIÓN

- 20.3** La función de un programa es encontrar el mayor de tres números. Desarrolle datos para realizar una prueba de caja blanca en esta sección del programa.

El código es:

```
int a, b, c;
int mayor;
if (a >= b) {
 if (a >= c) {
 mayor = a;
 }
 else {
 mayor = c;
 }
}
else {
 if (b >= c) {
 mayor = b;
 }
 else {
 mayor = c;
 }
}
```

## PRÁCTICA DE AUTOEVALUACIÓN

- 20.4** En un programa para jugar ajedrez, el jugador especifica el destino de un movimiento como un par de índices enteros: los números de fila y de columna. El programa verifica que el cuadro de destino sea válido; es decir, que no esté fuera del tablero. Desarrolle datos para realizar una prueba de caja blanca y comprobar que esta parte del programa funcione correctamente.

El código para esta parte del programa es:

```
if ((fila > 8) || (fila < 1)) {
 JOptionPane.showMessageDialog(null, "error");
}
if ((col > 8) || (col < 1)) {
 JOptionPane.showMessageDialog(null, "error");
}
```

## ● Inspecciones y recorridos

En la metodología de *inspección* o *recorrido* no utilizamos la computadora para tratar de erradicar los errores. En una inspección (o recorrido) alguien simplemente estudia el listado del programa (junto con la especificación) para tratar de ver los errores. Funciona mejor si la persona que realiza la inspección no es quien escribió el programa. Esto se debe a que las personas tienden a no ver sus propios errores. Entonces necesita pedirle a un amigo o colega que inspeccione su programa. Es extraordinario presenciar lo rápido que alguien más puede ver un error que nos ha estado venciendo durante horas.

Para inspeccionar un programa necesitamos:

- La especificación.
- El texto del programa en papel.

Para llevar a cabo una inspección, una de las estrategias es estudiar un método a la vez. Algunas de las comprobaciones son bastante mecánicas:

- Las variables inicializadas.
- Ciclos que se inicializan y terminan en forma correcta.
- Llamadas a métodos con los parámetros correctos.

Una comprobación más a fondo examina la lógica del programa. Simule ejecutar el método como si fuera una computadora, evitando seguir las llamadas de un método a otro método (por ello a esta metodología se le conoce como recorrido). Debe comprobar que:

- La lógica del método obtenga el propósito deseado.

Durante la inspección, puede comprobar que:

- Los nombres de los métodos y las variables sean representativos.
- La lógica sea clara y correcta.

Aunque el principal objetivo de una inspección no sea comprobar el estilo, una debilidad en cualquiera de estas áreas nos puede indicar un error.

La evidencia obtenida de los experimentos controlados nos sugiere que las inspecciones son una forma muy efectiva de encontrar errores. De hecho, las inspecciones son por lo menos tan efectivas para identificar errores como las pruebas en las que tenemos que ejecutar el programa (realizando el proceso de prueba).

## ● Paso a paso por instrucciones

Algunos sistemas de desarrollo de Java proveen un depurador con una herramienta para avanzar paso a paso por las instrucciones. Esto nos permite avanzar paso a paso por un programa, ejecutando sólo una instrucción a la vez. Cada vez que ejecutamos una instrucción, podemos ver la ruta de ejecución que ha tomado el programa. El depurador también nos permite mostrar (u observar) los valores de las variables. Es algo así como un recorrido estructurado automatizado.

En este tipo de prueba nos concentraremos en:

- Comprobar que la computadora ejecute la ruta esperada en todo el programa.
- Comprobar los valores de las variables a medida que el programa las va modificando, para verificar que se hayan modificado correctamente.

Mientras que el depurador por lo general se utiliza para la depuración (localizar un error), en esta técnica se utiliza para realizar pruebas (negar o confirmar la existencia de un error).

## ● Desarrollo incremental

Una de las metodologías para escribir un programa es escribirlo por completo en papel, introducirlo mediante el teclado y tratar de ejecutarlo. La palabra sobresaliente en este caso es “tratar”, ya que la mayoría de los programadores descubren que el amistoso compilador detectará muchos errores en su programa. Puede ser muy desalentador (en especial para los principiantes) ver tantos errores desplegados debido a un programa por el cual se esforzaron mucho. Una vez desterrados los errores de compilación, el programa por lo general exhibirá comportamientos extraños durante el (a veces extenso) periodo de depuración y prueba. Si se introducen mediante el teclado todas las partes de un programa a la vez para realizar el proceso de prueba, puede ser difícil localizar los errores. A esto se le denomina a veces desarrollo *big-bang*.

Una técnica útil alternativa para ayudarnos a evitar estas frustraciones es hacer las cosas parte por parte. Así, una alternativa al desarrollo big-bang es la programación parte por parte —a lo que por lo general se le denomina programación incremental. Los pasos son:

1. Diseñar y codificar la interfaz de usuario.
2. Escribir una pequeña parte del programa.
3. Introducirla mediante el teclado, corregir los errores de sintaxis, ejecutarla y depurarla.

4. Agregar una nueva y pequeña parte del programa.
5. Repetir desde el paso 2 hasta que el programa esté completo.

El truco aquí es identificar la parte del programa con la que podemos empezar y en qué orden debemos realizar las cosas. Tal vez la mejor metodología sea empezar por escribir el más simple de los métodos manejadores de eventos y después escribir los métodos que utilice este primer método. Después podemos escribir otro método manejador de eventos, y así sucesivamente.

La prueba incremental nos evita buscar una aguja en un pajar, ya que un error recién descubierto probablemente esté en el código que se acaba de incorporar.

## Principios de programación

No hay un método de prueba infalible que nos pueda asegurar que un programa esté libre de errores. La mejor metodología sería utilizar una combinación de los métodos de prueba —caja negra, caja blanca e inspección—, ya que la experiencia nos ha demostrado que encuentran distintos errores. Sin embargo, se requiere mucho tiempo para utilizar los tres métodos. En consecuencia, hay que ser muy precavidos y habilidosos para decidir qué tipo de prueba realizar y por cuánto tiempo se debe llevar a cabo. Es vital contar con una metodología sistemática.

El proceso de prueba es algo frustrante, ya que sabemos que no importa lo pacientes y sistemáticos que seamos, nunca podremos estar seguros de haber hecho lo suficiente. Para realizar las pruebas se requiere de una gran cantidad de paciencia, atención a los detalles y organización.

Escribir un programa es una experiencia constructiva, pero realizar pruebas es un proceso destructivo. Puede ser difícil tratar de demoler un objeto de orgullo que nos haya tomado horas crear; al encontrar un error necesitaremos más horas todavía para poder rectificar el problema. Por todo esto, es muy fácil comportarnos como un aveSTRUZ durante las pruebas y tratar de evitar los problemas.

## Resumen

- La prueba de un programa es una técnica que trata de establecer que un programa no contiene errores.
- Esta prueba no puede ser exhaustiva debido a que hay demasiados casos posibles.
- La prueba de la caja negra sólo utiliza la especificación para elegir los datos de prueba.
- En la prueba de la caja blanca debemos conocer cómo funciona el programa para poder elegir los datos de prueba.
- La inspección simplemente implica estudiar el listado del programa para ver si encontramos errores.
- Avanzar paso a paso por cada instrucción del código mediante un depurador puede ser una manera valiosa de probar un programa.
- El desarrollo incremental puede evitar las complejidades que surgen al desarrollar programas extensos.

## Ejercicios

- 20.1** Invierte datos de prueba para realizar las pruebas de la caja negra y de la caja blanca en el siguiente programa. La especificación es:

El programa recibe como entrada números enteros mediante un campo de texto y un botón. El programa muestra el mayor de los números introducidos hasta cierto punto.

Trate de no ver el texto del programa que le mostraremos a continuación, hasta después de que haya completado el diseño de los datos para la prueba de la caja negra.

A nivel de clase tenemos la declaración de una variable de instancia:

```
private int mayor = 0;
```

El código para manejar eventos es:

```
public void actionPerformed(ActionEvent event) {
 int número;
 número = Integer.parseInt(campoTexto.getText());
 if (número > mayor) {
 mayor = número;
 }
 resultado.setText("el mayor hasta ahora es " + Integer.toString(mayor));
}
```

- 20.2** Invierte datos de prueba para realizar las pruebas de la caja negra y de la caja blanca en el siguiente programa. La especificación se muestra a continuación. Trate de no ver el texto del programa que le mostramos a continuación, hasta después de que haya completado el diseño de los datos para la prueba de la caja negra.

El programa determina las primas de seguros para un día festivo, con base en la edad y el sexo (masculino o femenino) del cliente.

Para una mujer con edad  $\geq 18$  y  $\leq 30$ , la prima es de \$5.

Una mujer con edad  $\geq 31$  paga \$3.50.

Un hombre con edad  $\geq 18$  y  $\leq 35$  paga \$6.

Un hombre con edad  $\geq 36$  paga \$5.50.

Cualquier otro rango de edad o género es un error y se considera una prima de 0.

El código para este programa es:

```
public double calcPrima(double edad, String género) {
 double prima;

 if (género.equals("femenino")) {
 if ((edad >= 18) && (edad <= 30)) {
 prima = 5.0;
 }
 }
}
```

```
 else {
 if (edad >= 31) {
 prima = 3.50;
 }
 else {
 prima = 0;
 }
 }
 }
else {
 if (género.equals("masculino")) {
 if ((edad >= 18) && (edad <= 35)) {
 prima = 6.0;
 }
 else {
 if (edad >= 36) {
 prima = 5.5;
 }
 else {
 prima = 0;
 }
 }
 }
 else {
 prima = 0;
 }
}
return prima;
}
```

## Respuestas a las prácticas de autoevaluación

- 20.1** La especificación no indica qué debe ocurrir cuando surge una excepción. Hay varias posibilidades. La primera situación es si el usuario introduce datos que no sean un entero válido; por ejemplo, que introduzca una letra en vez de un dígito. La siguiente situación es si el usuario introduce un número mayor de 10000. La eventualidad final que podría surgir es si la suma de los números excede al tamaño del número que la computadora puede manejar. Si los enteros se representan como tipos `int` en el programa, este límite es enorme, pero podría darse el caso.

La especificación tampoco dice cómo señalar el final de los números.

**20.2** Un número de fila puede estar en tres particiones:

1. dentro del rango de 1 a 8;
2. menor que 1;
3. mayor que 8.

Si elegimos un valor representativo en cada partición (por decir 3, -3 y 11, respectivamente) y un conjunto similar de valores para los números de columna (por decir 5, -2 y 34), los datos de prueba serán:

| Número de prueba | Fila | Columna | Resultado |
|------------------|------|---------|-----------|
| 1                | 3    | 5       | Correcto  |
| 2                | -3   | 5       | Inválido  |
| 3                | 11   | 5       | Inválido  |
| 4                | 3    | -2      | Inválido  |
| 5                | -3   | -2      | Inválido  |
| 6                | 11   | -2      | Inválido  |
| 7                | 3    | 34      | Inválido  |
| 8                | -3   | 34      | Inválido  |
| 9                | 11   | 34      | Inválido  |

Ahora debemos considerar que los datos cerca del límite de las particiones son importantes y, por lo tanto, los agregamos a los datos de prueba para cada partición, de manera que tenemos lo siguiente:

1. Dentro del rango de 1 a 8 (por decir, 3).
2. Menor que 1 (por decir, -3).
3. Mayor que 8 (por decir, 11).
4. Valor de límite 1.
5. Valor de límite 8.
6. Valor de límite 0.
7. Valor de límite 9.

Lo que ahora nos da muchas más combinaciones que podemos usar como datos de prueba.

**20.3** Hay cuatro rutas a seguir en el programa, las cuales podemos recorrer con los siguientes datos de prueba:

| Número de prueba |   |   |   | Resultado |
|------------------|---|---|---|-----------|
| 1                | 3 | 2 | 1 | 3         |
| 2                | 3 | 2 | 5 | 5         |
| 3                | 2 | 3 | 1 | 3         |
| 4                | 2 | 3 | 5 | 5         |



*Respuestas a las prácticas de autoevaluación (continúa)*

- 20.4** Hay tres rutas a seguir en el extracto del programa, incluyendo la ruta en la que ninguna de las condiciones de las instrucciones `if` es verdadera. Pero cada uno de los mensajes de error se puede desencadenar con base en dos condiciones. Por lo tanto, los datos de prueba apropiados son:

| Número de prueba | Fila | Columna | Resultado |
|------------------|------|---------|-----------|
| 1                | 5    | 6       | Correcto  |
| 2                | 0    | 4       | Inválido  |
| 3                | 9    | 4       | Inválido  |
| 4                | 5    | 9       | Inválido  |
| 5                | 5    | 0       | Inválido  |

# CAPÍTULO **21**



## El proceso de depuración

En este capítulo conoceremos:

- Cómo depurar programas.
- Cómo usar un depurador.
- Los errores comunes.

### ● Introducción

Depurar es la acción de buscar los errores dentro de un programa y corregirlos. Todos los programas son propensos a contener errores la primera vez que son escritos. Un programa pasa por tres etapas a medida que es desarrollado:

1. Compilación.
2. Vinculación
3. Ejecución.

Ahora veremos cada uno de estos puntos de uno en uno.

### Compilación

Después de teclear un programa, por lo general es necesario invertir algo de tiempo en erradicar los errores de compilación. Un ejemplo común es la omisión del punto y coma. El compilador de Java realiza una serie exhaustiva de verificaciones en un programa, con lo cual expone muchos errores

que de lo contrario podrían persistir. A la larga, el programa se compila “limpiamente”. Una vez que el programa está libre de errores de compilación, por lo general se puede ejecutar —aun cuando tal vez no haga exactamente lo que deseamos.

## Vinculación

Todos los programas utilizan los métodos de biblioteca y algunos utilizan los métodos escritos por el programador. Un método se vincula sólo cuando alguien lo llama mientras el programa está en ejecución. Pero cuando se compila el programa, se verifica que todos los métodos que sean llamados existan y que el número y tipo de los parámetros coincidan. Por lo tanto, los errores de vinculación se detectan en tiempo de compilación.

## Ejecución

Un programa se ejecuta, pero es muy poco común que funcione según lo esperado. De hecho, es común que el programa falle en cierta forma o se comporte de una manera distinta a la esperada. Algunos errores se detectan automáticamente y se notifica al programador de ello —como al tratar de acceder a una parte de un arreglo que no existe (un error de índice). Otros son más sutiles y simplemente producen un comportamiento inesperado. No importa si tiene uno o muchos errores en su programa; debe llevar a cabo un proceso de depuración.

Más adelante en este capítulo veremos ejemplos de errores comunes que surgen al programar en Java.

El término *bug* se originó en los días de las computadoras de válvulas, cuando (según cuenta la historia) un enorme insecto (*bug*) quedó atrapado en los circuitos de una de las primeras computadoras, provocando que fallara. De aquí los términos en inglés “bug” y “debugging”, o depuración en español.

El problema con la depuración es que los síntomas de un error por lo general no son nada informativos. Por lo tanto, tenemos que recurrir al trabajo detectivesco para encontrar la causa. Es como ser un doctor: hay un síntoma, tenemos que encontrar la causa y después corregir el problema.

Una vez que eliminamos las fallas más obvias en un programa, es común empezar a realizar pruebas sistemáticas. Las pruebas constan de la ejecución repetida del programa con una variedad de datos como entrada; hablamos sobre este proceso en el capítulo 20. El objetivo de las pruebas es convencer al mundo de que el programa funciona en forma apropiada, pero por lo general revelan la existencia de más errores. Entonces es necesario depurar más el programa. De aquí que los procesos de prueba y depuración vayan de la mano.

## PRÁCTICA DE AUTOEVALUACIÓN

### 21.1 ¿Cuál es la diferencia entre depuración y prueba?

A muchos programadores les gusta el proceso de depuración; lo encuentran emocionante —algo así como ver una película de misterio en la que el villano se revela sólo hasta el último momento. Sin duda, el proceso de depuración (junto con las pruebas) requiere bastante tiempo. De hecho, a menudo la depuración tarda más que escribir el programa. No se preocupe porque el proceso de depuración sea tardado, ¡es algo normal!

## Cómo depurar sin un depurador

Un programa se ejecuta pero se comporta de forma inesperada. ¿Cómo podemos averiguar la causa del problema? La mayoría de los programas muestran algo en la pantalla, pero fuera de ello lo que hacen es invisible. Necesitamos algo así como lentes de rayos X para poder ver cómo se está comportando el programa. Ésta es la clave para una depuración exitosa: obtener información adicional sobre el programa en ejecución.

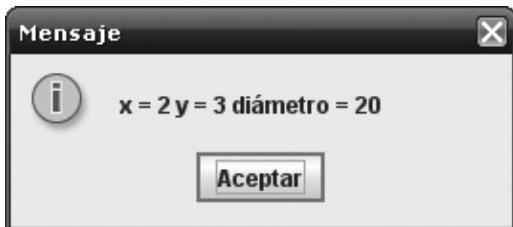
Una forma de obtener información adicional es insertar instrucciones de salida adicionales en el programa, de manera que muestre la información mientras se ejecuta. Una manera conveniente de mostrar información es a través de paneles de opciones. No olvide incluir la instrucción `import` apropiada para los paneles de opciones:

```
import javax.swing.JOptionPane;
```

Considere, por ejemplo, la clase `Globo` que se utilizó en el capítulo 9. Suponga que el objeto globo no aparece, o se muestra en el lugar equivocado. Para poder obtener información adicional, insertamos una instrucción para mostrar un panel de opción que revele la coordenada *x*, la coordenada *y* y el diámetro del globo. El código para hacer esto es:

```
JOptionPane.showMessageDialog(null,
 "x = " + x +
 " y = " + y +
 " diámetro = " + diámetro);
```

y la pantalla se muestra en la figura 21.1.



**Figura 21.1** Pantalla que se utiliza para la depuración.

Ahora podemos ver si esta información es correcta o si nos proporciona información útil sobre el error.

El truco es elegir los mejores puntos en el programa en donde podemos colocar instrucciones para mostrar paneles de opción. Si insertamos demasiadas instrucciones de rastreo en el programa, habrá mucha información irrelevante y molesta. En general, los lugares adecuados son:

- Justo antes de llamar a un método (para verificar que los parámetros sean correctos) o justo al inicio del método.
- Justo después de llamar a un método (para verificar que el método haya realizado su trabajo correctamente) o justo al final del método.

## Cómo usar un depurador

En este libro evitamos a propósito hacer referencia a cualquier paquete de desarrollo específico. El sistema de Java que usted utiliza puede tener un “depurador”. Un depurador es un programa que le ayuda a depurar su programa. Se ejecuta junto con su programa y permite inspeccionar el progreso del mismo. Un depurador cuenta con varias herramientas, pero por desgracia los depuradores no están estandarizados; todos son distintos y por lo tanto no podemos explicar todas las distintas herramientas con detalle. Algunos de los depuradores disponibles son bastante primitivos. Por ejemplo, el jdb (depurador de java) está basado en texto y orientado a la línea de comandos, con ascendencia de Unix. También hay buenos depuradores disponibles, los cuales forman parte de un IDE.

Sin embargo, explicaremos los principios sobre el uso de un buen depurador interactivo e integrado. Imagine tres ventanas en la pantalla, todas visibles al mismo tiempo:

- La ventana creada por el programa en ejecución.
- La pantalla del código fuente del programa de Java.
- Una pantalla de los nombres de las variables seleccionadas junto con sus valores actuales.

## Puntos de interrupción

Mediante el depurador, el programador puede colocar un *punto de interrupción* en el programa. Un punto de interrupción es un lugar en el programa en donde se detiene la ejecución. Para insertar uno hay que apuntar a una línea en el texto del programa con el cursor y seleccionar una opción del menú. Después se ejecuta el programa. Al llegar a un punto de interrupción el programa se congela, un puntero resalta la posición del punto de interrupción en el texto del programa y el depurador muestra los valores de las variables en ese momento en la ejecución del programa. Cualquier discrepancia entre su valor actual y el que deberían tener nos proporciona información valiosa para la depuración.

## Cómo recorrer el programa paso a paso

El depurador también nos permite ejecutar un programa una línea a la vez; a esto se le conoce como *recorrido paso a paso*. Podemos seguir el progreso del programa y verificar que se comporte según lo esperado. Cuando no lo haga, sabremos que estamos a punto de localizar un error.

Podemos asumir con seguridad que los métodos de biblioteca están libres de errores, por lo que no es conveniente desperdiciar tiempo recorriendo estos métodos paso a paso. Por lo general, los depuradores proveen una opción para ejecutar los métodos seleccionados a la velocidad normal.

Aunque podemos llegar a divertirnos mucho con un depurador, la desventaja es que se puede requerir mucho tiempo. A continuación le presentamos una manera productiva de utilizar un depurador:

1. Haga una hipótesis sobre la posible ubicación del error con base en los síntomas del mismo. Tal vez pueda predecir que el error se encuentra dentro de dos o tres métodos cualesquiera.
2. Coloque puntos de interrupción a la entrada y salida de los métodos sospechosos.
3. Ejecute el programa. Cuando éste se detenga en la entrada de un método, inspeccione los valores de los parámetros. En la salida, inspeccione el valor de retorno y los valores de las

variables de instancia importantes. Con esto podrá identificar el método dentro del cual está el error.

4. Ejecute el programa de nuevo y deténgase en la entrada del método con el error. Avance paso a paso por este método hasta que vea la discrepancia entre lo esperado y la realidad.
5. Así habrá encontrado el error.

Los programadores de Java cometan ciertos errores comunes. A continuación le mostraremos una lista de ellos. Vale la pena revisar cualquier programa sospechoso en busca de estos errores.

## ● Errores comunes – errores de compilación

El compilador de Java realiza un proceso exhaustivo de verificación en un programa mientras lo compila en código de bytes. Esto es parte del proceso de asegurar que los programas de Java sean robustos. Un error que se atrapa en tiempo de compilación se puede corregir con facilidad, pero si se deja sin detectar hasta el tiempo de ejecución, se requerirá mucho tiempo de depuración. Por lo tanto, aunque los errores en tiempo de compilación pueden ser molestos, vale la pena detectarlos a tiempo.

### Punto y comas

Éste es el error más común de compilación: omitir los signos de punto y coma.

### Instrucciones if

Es fácil olvidar que las instrucciones `if` deben tener paréntesis para encerrar la condición, como en el siguiente ejemplo:

```
if (a == b) etc.
```

### Llaves

Un error común es omitir una llave (`{` o `}`) o colocar una en el lugar equivocado. El buen uso consistente de la sangría le puede ayudar a detectar y corregir este problema.

### Llamadas a métodos con cero parámetros

Si un método llamado `hacerEso` no recibe parámetros, es fácil escribir una llamada al método en la que se omitan los paréntesis:

```
objeto.hacerEso; en vez de objeto.hacerEso();
```

Lo anterior se considera como un intento de acceder a una variable llamada `hacerEso` dentro del objeto.

### Llamadas a métodos

Un error común es escribir mal el nombre de un método. Otro error es usar el número incorrecto de parámetros o el tipo incorrecto de un parámetro.

## Llamadas a métodos – omitir import

Falta una instrucción `import` o ésta especifica el paquete incorrecto. Esto ocurre a menudo si utilizamos componentes de Swing como los controles deslizables.

## Llamadas a métodos – omitir extends

En el encabezado de la clase falta la palabra `extends`, seguida del nombre de la clase relevante. La mayoría de los programas de este libro extienden a la clase `JFrame`.

## ● Errores comunes – errores en tiempo de ejecución

Los errores en tiempo de ejecución son aquellos que ocurren a medida que se ejecuta el programa, pero son atrapados por el sistema en tiempo de ejecución. De nuevo, esto es parte de las medidas diseñadas para asegurar que los programas sean robustos. Sin la verificación en tiempo de ejecución, un programa con error podría actuar como un toro en una tienda de porcelana china. Al detectar un error en tiempo de ejecución, aparece un mensaje de error y se detiene el programa.

## Índices de arreglos

Suponga que declaramos un arreglo de la siguiente manera:

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

y a continuación el siguiente ciclo `for` tratará de utilizar incorrectamente el 10º elemento del arreglo:

```
for (int s = 0; s <= 10; s++) {
 tabla[s] = 0;
}
```

Cuando esto ocurre, el programa se detiene y se lanza una excepción `ArrayIndexOutOfBoundsException`.

## Excepciones aritméticas

Si un programa intenta dividir un entero entre 0, el programa se detendrá y aparecerá un mensaje de error para describir una excepción `ArithmaticException`. Es bastante fácil dejar que esto ocurra de manera inadvertida; por ejemplo, en un programa que contenga este fragmento:

```
int a, b, c, d;
a = b/(c-d);
```

La aritmética de punto flotante no ocasiona excepciones como ésta; en vez de ello, el programa continúa y el resultado de dividir entre 0 es infinito (vea el capítulo 11).

## Excepción de apuntador nulo

Si declara un objeto como el siguiente:

```
JButton botón;
```

y después lo utiliza sin crear una instancia del objeto (mediante `new`):

```
add(botón);
```

se producirá una excepción `NullPointerException` para indicar que está tratando de usar un objeto que no se ha creado todavía.

## ● Errores comunes – errores lógicos

Los errores lógicos son los más difíciles de encontrar, ya que dependen de la manera en que funciona el programa individual. Por lo tanto, dichos errores no se pueden detectar en forma automática. Sin embargo, algunos errores específicos son comunes.

### Inicialización

Es fácil fracasar al inicializar una variable en forma apropiada. En Java, todas las variables se inicializan de manera automática con cierto valor definido (por ejemplo, los enteros se inicializan con cero de manera automática), aunque tal vez no sea el valor requerido. Algunos compiladores marcan las variables que se utilizan sin inicializar. Al principio los arreglos están llenos de basura, a menos que se inicialicen de alguna manera. La mejor práctica es inicializar todos los datos de manera explícita.

### Eventos

Usted hace clic en un botón pero no ocurre nada. Tal vez no haya proporcionado bien el manejo para un evento.

## ● Errores comunes – malentender el lenguaje

Si el programador no comprende completamente cómo usar Java en forma apropiada, los programas que parezcan perfectamente sanos no funcionarán de la manera esperada. He aquí algunos casos comunes.

### Llaves

Puede omitir los paréntesis en las instrucciones `if` y en los ciclos, siempre y cuando sólo haya una instrucción involucrada. Esto puede producir código como el siguiente:

```
if (a > b)
 x = x + a;
 y = y + b;
 z = z + c;
```

La distribución sugiere que se deben ejecutar dos instrucciones si la condición es verdadera. Pero se omitieron las llaves, por lo que sólo se ejecutará una instrucción si la condición es verdadera. La sangría puede provocar malentendidos. Para que funcione como lo sugiere la sangría, inserte llaves como se muestra a continuación:

```

if (a > b) {
 x = x + a;
 y = y + b;
}
z = z + c;

```

## Igualdad parte 1

Al usar una instrucción `if` para probar si dos cosas son iguales, lo siguiente siempre está equivocado:

```
if (a = b) ...
```

Esto se debe a que está utilizando el operador de asignación (`=`) en vez del operador de comparación (`==`). Por desgracia, el compilador no marca este error.

## Igualdad parte 2

Es tentador escribir instrucciones tales como:

```
if (s == "abc") ...
```

en donde `s` es una cadena de texto. Esto se compilará correctamente, pero si el programador desea evaluar si `s` es igual a `"abc"`, no funcionará como se espera. Para lograr esto se requiere la siguiente evaluación:

```
if (s.equals("abc")) ...
```

la cual utiliza el método `equals` de la clase `String`.

La primera instrucción `if` es significativa, pero evalúa si la cadena `s` y la literal `"abc"` son en realidad el mismo objeto, lo cual no es así. La cadena `s` es un objeto que el programa creó a través de `new`:

```
String s = new String();
```

mientras que la cadena `"abc"` es una cadena completamente distinta, creada de manera automática por el compilador. Entonces estas dos cadenas no son, y nunca podrán ser, el mismo objeto (sus valores podrían ser iguales, pero no son el mismo objeto).

Este ejemplo utiliza cadenas, pero la misma idea se aplica cuando se compara la igualdad entre dos objetos; el operador `==` casi siempre es incorrecto, por lo que debemos usar en su defecto el método `equals` del objeto. La única ocasión en que está bien usar `==` es para comparar enteros y otros tipos primitivos.

## Ciclos infinitos

Es fácil equivocarse y colocar un punto y coma justo después de una instrucción `while`, como en el siguiente ejemplo:

```

int a;
while (a == 0);

```

Este error no lo detecta el compilador; provocará que el programa itere en forma indefinida.

## Resumen

- Depurar significa encontrar errores (bugs) en un programa y corregirlos.
- Algunos sistemas de desarrollo de Java proveen un programa “depurador” que puede ser útil.
- Un punto de interrupción es un lugar en donde el programa se detiene en forma temporal para poder inspeccionar los valores de las variables.
- Avanzar paso a paso significa observar el flujo de ejecución del programa, una instrucción a la vez.

## Respuesta a la práctica de autoevaluación

- 21.1** Probar consiste en tratar de demostrar que un programa está libre de errores. Si sabemos que existe un error (como resultado de la prueba), depurar consiste en tratar de localizar ese error.

# CAPÍTULO **22**



## Hilos

En este capítulo conoceremos:

- Los conceptos de hilo y multihilo.
- Cómo crear hilos.
- Cómo iniciar y eliminar hilos.
- Los distintos estados de un hilo.

### ● Introducción

Empezaremos por analizar varias situaciones, algunas del mundo real y otras en las que se involucran las computadoras.

Nuestro primer ejemplo es real y humano: un grupo de personas que compran una comida. Suponga que tres estudiantes comparten una casa. Deciden hacer de comer y se dividen la tarea de ir de compras:

- Uno compra la carne (no son vegetarianos).
- Uno compra los vegetales.
- Uno compra la cerveza.

Y hacen estas cosas al mismo tiempo. Pero debemos tener en cuenta que en un momento dado deberán sincronizar sus actividades para poder preparar la comida y que todos se sienten a comer al mismo tiempo.

Un grupo de músicos en una orquesta tocan distintos instrumentos al mismo tiempo y leen sus distintas partes en una partitura musical. La partitura indica a cada uno de ellos lo que deben hacer y les ayuda a sincronizar sus notas.

Ahora considere una computadora que controla una planta industrial, como una fábrica de panecillos. La planta consiste en una variedad de equipo, como hornos, básculas, válvulas y bombas. Estos dispositivos se deben controlar al mismo tiempo.

Cuando usamos una PC, le podemos pedir que haga varias cosas a la vez: imprimir un archivo, editar un archivo distinto, mostrar la hora y recibir correo electrónico. En realidad, una sola computadora puede hacer sólo una cosa a la vez, pero como trabaja con tanta rapidez, puede compartir su tiempo entre varias actividades. Hace esto tan rápido que da la impresión de que lo está haciendo todo a la vez.

Cuando un programa de Java hace varias cosas a la vez, se le denomina *multihilo* (otros términos que se utilizan para multihilo son paralelismo y concurrencia). Dicho programa puede hacer varias cosas aparentemente al mismo tiempo: mostrar varias animaciones, reproducir sonidos, permitir que el usuario interactúe. A cualquiera de las actividades que realiza se le conoce como *hilo*. En este capítulo veremos cómo escribir hilos en Java.

Hemos visto que un programa es una serie de instrucciones para la computadora, que por lo general obedece en secuencia. La secuencia se desvía mediante llamadas a métodos, ciclos e instrucciones de selección (`if`), pero de todas formas sigue siendo una sola ruta que avanza por el programa. Podemos usar un lápiz para simular la ejecución de un programa, siguiendo la ruta a través del mismo. En la programación multihilo hay dos o más rutas en ejecución. Para seguir las rutas de ejecución ahora necesitamos varios lápices, uno para cada hilo. Ahora, si recordamos que una computadora sólo es capaz de realizar una cosa a la vez, podemos deducir entonces que divide su tiempo disponible entre los distintos hilos para dar la impresión de que todos se ejecutan al mismo tiempo.

Todos los programas que hemos visto hasta ahora en el libro consisten en un solo hilo. La computadora sigue una sola ruta a través del programa. Algunas veces el programa no hace nada mientras espera un evento. Sin embargo, cuando surge un evento hay de nuevo una sola ruta de ejecución a través del programa, hasta que se completa el manejo del evento.

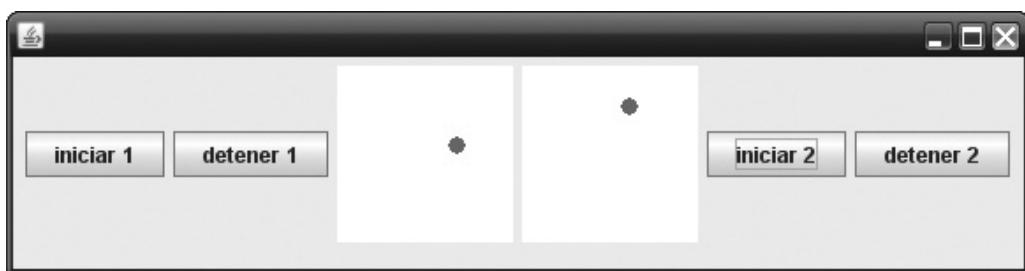
Pero suponga que el programa tiene que llevar a cabo una tarea que requiere cierto tiempo. Algunos ejemplos son:

- Copiar un archivo.
- Descargar una página Web.
- Mostrar una animación.

En casos como éstos, el hilo individual se preocupa por la tarea que consume más tiempo. La consecuencia es que el programa no puede responder a los eventos del usuario. Ningún botón de la ventana funcionará. Incluso los iconos de la ventana (para minimizar y mover) no funcionan. Un programa como éste necesita una reestructuración para que utilice multihilos. Un hilo es la GUI, lista para responder en cualquier momento a los eventos. Un segundo hilo se encarga de la actividad que consume mucho tiempo.

## Hilos

En este capítulo vamos a estudiar un programa que hace que una pelota rebote alrededor de un panel y que otra pelota rebote alrededor de un segundo panel (figura 22.1): un programa de animación. El programa cuenta con un botón etiquetado como “iniciar” para que una pelota empiece a rebotar y un segundo botón para iniciar la otra pelota.



**Figura 22.1** Pelotas que rebotan.

Para asegurar que el programa pueda responder a los eventos de los usuarios en cualquier momento, creamos un hilo de interfaz de usuario. Para implementar las dos animaciones de las pelotas, creamos un hilo para cada una. Por lo tanto, hay tres hilos:

1. La interfaz de usuario (también tiene la tarea de iniciar los otros hilos).
2. Un hilo para la pelota 1.
3. Un hilo para la pelota 2.

Cada uno de estos hilos es un objeto distinto y los tres se ejecutan en conjunto, compartiendo el procesador.

Hay dos clases:

1. La interfaz de usuario.
2. La clase que describe a una pelota.

Este programa funciona muy bien. Al oprimir uno de los botones “iniciar”, se crea una pelota y ésta rebota en el panel hasta que se oprime el botón “detener”. El hilo de interfaz de usuario siempre está atento a las acciones del usuario (hacer clic en un botón). Los hilos de las pelotas se ejecutan de manera independiente. Cada pelota se muestra en pantalla a través de su propio hilo independiente.

La función de la animación es mover un objeto, dibujar su imagen, esperar un breve intervalo de tiempo, eliminar la imagen y volver a hacer todo el proceso repetidas veces. Para esperar, el programa hace una llamada al método de biblioteca `Thread.sleep`, el cual tiene un parámetro que especifica cuánto tiempo debe esperar el programa. El parámetro que se pasa a `sleep` es el tiempo en milisegundos que el programa desea permanecer inactivo. Al utilizar el método `sleep`, Java insiste que el programador proporcione un manejador de excepciones, como se muestra en el código (en el capítulo 16 vimos las excepciones). Cuando un hilo está inactivo, algún otro hilo podría tratar de interrumpir su inactividad. Esta situación se denomina excepción tipo `InterruptedException`. En este programa específico se puede ignorar sin problemas (debido a que no hay otros hilos que podrían tratar de hacer esto).

He aquí el programa completo:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Pelotas extends JFrame implements ActionListener {
```

```
private JButton iniciar1, iniciar2, detener1, detener2;
private JPanel panel1, panel2;
private Pelota pelota1, pelota2;

public static void main(String[] args) {
 Pelotas marco = new Pelotas();
 marco.setSize(500,150);
 marco.crearGUI();
 marco.setVisible(true);
}

private void crearGUI() {
 setDefaultCloseOperation(EXIT_ON_CLOSE);
 Container ventana = getContentPane();
 ventana.setLayout(new FlowLayout());

 iniciar1 = new JButton("iniciar 1");
 ventana.add(iniciar1);
 iniciar1.addActionListener(this);

 detener1 = new JButton("detener 1");
 ventana.add(detener1);
 detener1.addActionListener(this);

 panel1 = new JPanel();
 panel1.setPreferredSize(new Dimension(100, 100));
 panel1.setBackground(Color.white);
 ventana.add(panel1);

 panel2 = new JPanel();
 panel2.setPreferredSize(new Dimension(100, 100));
 panel2.setBackground(Color.white);
 ventana.add(panel2);

 iniciar2 = new JButton("iniciar 2");
 ventana.add(iniciar2);
 iniciar2.addActionListener(this);

 detener2 = new JButton("detener 2");
 ventana.add(detener2);
 detener2.addActionListener(this);
}

public void actionPerformed(ActionEvent event) {
 if (event.getSource() == iniciar1) {
 pelota1 = new Pelota(panel1);
 pelota1.start();
 }
 if (event.getSource() == iniciar2) {
 pelota2 = new Pelota(panel2);
 pelota2.start();
 }
 if (event.getSource() == detener1) {
 pelota1.detenerPorFavor();
 }
}
```

```
 }
 if (event.getSource() == detener2) {
 pelota2.detenerPorFavor();
 }
 }
}

import java.awt.*;
import javax.swing.*;

public class Pelota extends Thread {
 private JPanel panel;
 private int x = 7, cambioX = 7;
 private int y = 0, cambioY = 2;
 private final int diámetro = 10;
 private final int anchura = 100, altura = 100;
 boolean seguirRebotando;

 public Pelota(JPanel elPanel) {
 panel = elPanel;
 }

 public void run() {
 seguirRebotando = true;
 while (seguirRebotando) {
 mover();
 rebotar();
 dibujar();
 retrasar();
 eliminar();
 }
 }

 private void mover() {
 x = x + cambioX;
 y = y + cambioY;
 }

 private void rebotar() {
 if (x <= 0 || x >= anchura) {
 cambioX = -cambioX;
 }
 if (y <= 0 || y >= altura) {
 cambioY = -cambioY;
 }
 }

 private void retrasar() {
 try {
 Thread.sleep(50);
 }
 catch (InterruptedException e) {
 return;
 }
 }
}
```

```

private void dibujar() {
 Graphics papel = panel.getGraphics();
 papel.setColor(Color.red);
 papel.fillOval(x, y, diámetro, diámetro);
}

private void eliminar() {
 Graphics papel = panel.getGraphics();
 papel.setColor(Color.white);
 papel.fillOval (x, y, diámetro, diámetro);
}

public void detenerPorFavor() {
 seguirRebotando = false;
}
}

```



## Cómo iniciar un hilo

En el programa de la pelota que rebota, el hilo de la interfaz de usuario es (como siempre) una extensión de la clase de biblioteca `JFrame`. Este hilo crea un nuevo objeto pelota de la misma forma en que se crea cualquier otro objeto, utilizando `new`:

```
Pelota pelota1 = new Pelota(panel1);
```

El objeto interfaz de usuario también pide al sistema de Java que ejecute el hilo a través del método de biblioteca `start`:

```
pelota1.start();
```

La segunda clase (`Pelota`) es un hilo distinto. Un hilo siempre debe:

1. Extender (heredar de) la clase de biblioteca `Thread`.
2. Proveer un método llamado `run`, que el sistema de Java invoca para poner el hilo en ejecución.

Entonces, al llamar a `start` se inicia un hilo, lo cual a su vez provoca una llamada a `run`. Tan pronto como ocurre esto, el hilo original (el que llamó a `start`) continúa y el nuevo hilo se ejecuta. Por lo tanto se ejecutan juntos, en paralelo.

## PRÁCTICA DE AUTOEVALUACIÓN

**22.1** ¿Cuántos hilos se requieren para cada una de las siguientes actividades?:

1. Antes de hacer clic en un botón.
2. Despues de hacer clic en “iniciar pelota 1”.
3. Despues de hacer clic en “iniciar pelota 2”.

## Muerte de un hilo

En el programa de la pelota que rebota, una pelota sigue rebotando hasta que la detengamos haciendo clic en un botón. La manera de terminar un hilo de una pelota que rebota es mediante una bandera `boolean`, a la cual llamaremos `seguirRebotando`. Al principio esta variable es `true` cuando inicia el hilo, pero se establece en `false` cuando el usuario oprime el botón “detener”. El ciclo en el objeto pelota es:

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

lo cual hará que una pelota rebote mientras la variable `boolean` sea `true` y que se detenga cuando el valor sea `false`.

En el hilo de la interfaz de usuario proveemos un botón etiquetado como “detener1”, de la forma usual. La parte del manejador de eventos que se requiere para manejar este botón “detener1” es:

```
if (event.getSource() == detener1) {
 pelota1.detenerPorFavor();
}
```

y el método en la clase `Pelota` es:

```
public void detenerPorFavor() {
 seguirRebotando = false;
}
```

Cuando el hilo `pelota1` termina de iterar, sale del método `run` y, por lo tanto, muere. Ésta es la forma normal en que muere un hilo.

## PRÁCTICA DE AUTOEVALUACIÓN

**22.2** ¿Cuántos hilos hay detrás de “iniciar pelota 1”, “iniciar pelota 2” y después “detener pelota 1”?

## join

Al usar multihilos tal vez sea conveniente hacer que un hilo espere hasta que otro haya muerto. Esto se hace con el método `join`. Por ejemplo, suponga que `pelota1` hace lo siguiente:

```
pelota2().join;
```

Entonces, `pelota1` espera hasta que el hilo `pelota2` haya muerto.

## El estado de un hilo

Puede ser útil comprender en qué consisten los estados de un hilo. Un hilo puede estar en uno de los siguientes estados:

- **new:** se acaba de crear con el operador `new`, justo igual que cualquier otro objeto, pero aún no se ha establecido como ejecutable (runnable) por medio del método `start`.
- **running:** en este momento está ejecutando instrucciones en la computadora.
- **runnable:** le gustaría ejecutarse, puede ejecutarse pero algún otro hilo se está ejecutando.
- **blocked:** por alguna razón el hilo no puede proceder sino hasta que algo ocurra. Un ejemplo típico es cuando un hilo llama a `sleep` y queda suspendido por varios milisegundos.
- **dead:** el hilo salió del método `run` de manera normal.

Un hilo puede averiguar el estado de otro hilo, para lo cual llama al método de biblioteca `isAlive`. Este método devuelve `true` si el hilo es ejecutable o está bloqueado. Devuelve `false` si el hilo es nuevo o está muerto. Por ejemplo:

```
if (pelota1.isAlive()) {
 estado.setText("la pelota 1 está viva");
}
```

## PRÁCTICA DE AUTOEVALUACIÓN

- 22.3 Modifique el programa de la pelota que rebota de manera que al hacer clic en un botón “ir”, éste no funcione cuando el hilo de la pelota ya esté vivo.

## ● Planificación de procesos, prioridades de hilos y `yield`

El sistema de Java tiene la tarea de compartir el procesador individual (o algunas veces multiprocesador) entre los hilos de un programa. A menudo se da el caso de que hay varios hilos que son ejecutables (están listos y pueden ejecutarse cuando el procesador esté disponible). El programador debe elegir entre ellos. Los distintos planificadores de procesos en distintas máquinas y sistemas de Java pueden trabajar de maneras diferentes. No hay garantía de que se proporcione una calendarización específica. El programador debe tener cuidado de no hacer ninguna suposición en cuanto al momento en que se seleccionará un hilo específico para ejecutarlo.

El sistema de Java proporciona de manera predeterminada la misma prioridad a todos los hilos que se crean. A partir de ahí el sistema de Java compartirá el procesador en forma equitativa entre los hilos activos. Aún así, es fácil que un hilo acapare el procesador y desconecte a los demás hilos. Por lo tanto, cualquier hilo con una conciencia social debería llamar al método de biblioteca `yield` de la siguiente manera::

```
yield();
```

esto se hace cuando el programador estima que el hilo puede ser avaricioso. Si otros hilos son ejecutables, reciben una parte del tiempo del procesador. Pero si no hay otros hilos en el estado ejecutable, el mismo hilo continúa.

## Principios de programación

La principal motivación de la programación multihilos es la necesidad de hacer dos o más cosas al mismo tiempo. En especial, los programas necesitan una interfaz de usuario que siempre responda. Por lo tanto, un programa se divide en partes que se ejecuten al mismo tiempo.

La concurrencia es muy común, tanto en la vida real como en los sistemas de cómputo. A una actividad paralela dentro de una computadora se le conoce como hilo, tarea o algunas veces proceso (Java utiliza el primer término). El código de cada hilo individual se ve como un programa secuencial normal. El sistema de Java comparte el tiempo del procesador entre los hilos de tal forma que parezca como si todos se ejecutaran en paralelo. A esto algunas veces se le conoce como concurrencia aparente. Si dos personas están colaborando en una comida, o si un sistema de cómputo consta de varios procesadores, entonces hay una concurrencia real. Algunos científicos de computación usan el término “paralelismo” para designar a la concurrencia aparente y a la concurrencia real simplemente le dicen “concurrencia”. Sin importar la terminología que adoptemos, una característica esencial de la situación es que el paralelismo está explícitamente en las manos del programador.

Hemos visto cómo crear hilos que se ejecutan en paralelo, compartiendo el procesador. Una característica clave de este ejemplo es que los hilos no interactúan ni se comunican con los otros hilos. Una vez creados, son hilos independientes. Éste es un escenario común y muy útil en la programación multihilo.

Hay escenarios multihilo más complejos. Un escenario es cuando los hilos se comunican entre sí para realizar cierta tarea. Uno necesita decirle a otro que algo ocurrió. Otro escenario es cuando dos o más hilos necesitan acceder a ciertos datos compartidos, como un campo de texto. Existe el peligro de que interfieran entre sí y que los datos no sean confiables. Estas situaciones introducen complicaciones que están más allá del alcance de este libro.

## Resumen

Ahora sabe todo lo que necesita para escribir programas que hagan muchas cosas al mismo tiempo. Puede mostrar un reloj, mostrar una animación y jugar un juego; todo al mismo tiempo. Necesita crear un hilo para cada una de estas actividades paralelas, como vimos antes.

Es común en los programas de Java que dos o más hilos se ejecuten en paralelo. Cada hilo se comporta como un programa secuencial normal. El calendarizador comparte el tiempo del procesador entre los hilos que están listos para ejecutarse. El método `start` se utiliza para establecer un nuevo proceso en ejecución. Una clase de hilo se debe declarar de manera que extienda a la clase `Thread`; además debe proveer un método `run` que se ejecute al iniciar el hilo.

Un hilo muere cuando sale de su método `run`.

Algunas veces podemos evitar el uso de hilos. Por ejemplo, en un programa de animación como el que utilizamos en este capítulo, una alternativa es usar la clase de biblioteca `Timer`. Podemos usar esta clase para lanzar eventos a intervalos preestablecidos. Estos eventos se pueden usar para mover y volver a dibujar la imagen en vez de usar el método `sleep` para suspender un hilo. Sin embargo, no siempre se puede utilizar un temporizador en lugar de los hilos.

## Ejercicios

**22.1 Pelotas que rebotan** Extienda y modifique el programa de manera que haya tres pelotas que reboten; todas deben rebotar dentro del mismo panel.

**22.2 Reloj** Escriba un programa para mostrar un reloj digital. El reloj debe mostrar las horas, minutos y segundos, cada uno en su propio campo de texto. Un botón “iniciar” debe iniciar el reloj y un botón “detener” debe detenerlo.

Cree un hilo para la GUI y un segundo hilo para el reloj. El hilo del reloj debe obtener la hora (vea más abajo) y después debe permanecer inactivo por 100 milisegundos, lo cual equivale a 1/10 de segundo.

Para obtener el tiempo, el programa usa la clase de biblioteca **Calendar** dentro del paquete **Java.util**. Para crear una instancia de esta clase, haga lo siguiente:

```
Calendar calendario = Calendar.getInstance();
```

Después, para obtener el valor de la parte de la hora actual correspondiente a los segundos, haga esto:

```
int segundos = calendario.get(Calendar.SECOND);
```

y se puede hacer de manera similar para las partes de la hora actual correspondientes a las horas (**HOUR**), minutos (**MINUTE**) y milisegundos (**MILLISECOND**).

**22.3 Campo de texto desplazable** Éste es un clásico programa multihilos. Escriba un programa que muestre un campo de texto, el cual debe contener texto que se desplace de derecha a izquierda. Un botón debe iniciar el desplazamiento y otro debe detenerlo. Necesitará usar el método **substring** para manipulación de cadenas (vea el capítulo 15).

## Respuestas a las prácticas de autoevaluación

**22.1** 1. Uno, el hilo de la interfaz de usuario.

2. Dos.
3. Tres.

**22.2** Dos, el hilo de la interfaz de usuario y un hilo de pelota.

```
22.3 if ((event.getSource() == iniciar1) && !(pelota1.isAlive())) {
 pelota1 = new Pelota(panel1);
 pelota1.start();
}
```

# CAPÍTULO **23**



## Interfaces

En este capítulo conoceremos:

- Cómo utilizar interfaces para describir la estructura de un programa.
- Cómo utilizar interfaces para asegurar la interoperabilidad de las clases dentro de un programa.
- Cómo una clase puede implementar varias interfaces.
- Las diferencias entre las interfaces y las clases abstractas.

### ● Introducción

Java posee una notación para describir la apariencia externa de una clase, a la cual llamamos *interfaz*. La descripción de una interfaz es semejante a la de una clase, sólo que sin los cuerpos de los métodos. No debemos confundir el uso que hacemos aquí de la palabra “interfaz” con la misma palabra que se utiliza en el término interfaz gráfica de usuario (GUI). Las interfaces tienen dos usos:

- En el diseño.
- Para promover la interoperabilidad.

### ● Interfaces para el diseño

Con frecuencia hacemos hincapié sobre la importancia del diseño durante la planeación inicial de un programa. Para ello hay que diseñar todas las clases del mismo. Una forma de documentar dicho diseño es escribir en español una especificación de los nombres de las clases y sus métodos. Pero también es posible escribir esta descripción en Java. Por ejemplo, la interfaz para la clase `Globo`, que se utiliza a menudo en este libro, es:

```
public interface GloboInterfaz {
 void cambiarTamaño(int nuevoDiámetro);
 int getX();
 void setX(int x);
 void mostrar(Graphics papel);
}
```

Cabe mencionar que omitimos la palabra **class** en la descripción de una interfaz. Además, los métodos no se declaran como **public** (ni como cualquier otra cosa) ya que son implícitamente **public**.

En una interfaz sólo describimos los nombres de los métodos y sus parámetros, mientras que omitimos los cuerpos de los métodos. Una interfaz describe a una clase, pero no dice cómo debemos implementar los métodos y elementos de datos. Por ende, sólo describe los servicios que proporciona la clase —representa la apariencia externa de una clase como la ven sus usuarios (o de un objeto que se crea como instancia de esa clase). En consecuencia, también nos dice qué debe proporcionar la persona que implemente esa clase.

## PRÁCTICA DE AUTOEVALUACIÓN

- 23.1 Agregue instrucciones a la interfaz **GloboInterfaz** que describa métodos para acceder al valor de *y* y modificarlo.

Podemos compilar una interfaz junto con cualquier otra clase, pero en definitiva no podemos ejecutarla. Sin embargo, alguien que planee **usar** una clase puede compilar el programa junto con la interfaz y comprobar así que se utilice en forma correcta. Cualquiera que haya escrito un programa en Java sabe que el compilador vigila con mucha cautela en busca de errores que, de pasar inadvertidos, podrían ocasionar problemas maliciosos a la hora de ejecutar el programa. Por lo tanto, cualquier comprobación que se realice en tiempo de compilación es algo que bien vale la pena.

Una persona que implementa una interfaz puede especificar en el encabezado de la clase que va a implementar cierta interfaz específica. En un ejemplo anterior escribimos una interfaz para la clase **Globo**. Ahora vamos a escribir la clase **Globo** para relacionarla con la interfaz:

```
import java.awt.Graphics;

public class Globo implements GloboInterfaz {

 private int diámetro, x, y;

 public void cambiarTamaño(int nuevoDiámetro)
 diámetro = nuevoDiámetro;
 }

 public int getX() {
 return x;
 }
}
```

```

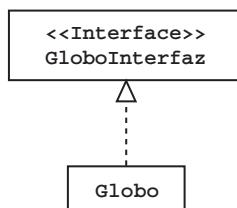
public void setX(int nuevaX) {
 x = nuevaX;
}

public void mostrar(Graphics papel) {
 papel.setColor(Color.black);
 papel.fillOval(x, y, diámetro, diámetro);
}
}

```

Para describir la clase como un todo decimos que implementa a la interfaz **GloboInterfaz**. El compilador comprueba entonces que esta clase esté escrita de manera que cumpla con la declaración de la interfaz; es decir, que proporcione los métodos **cambiarTamaño**, **setX**, **getX** y **mostrar**, junto con sus parámetros apropiados. La regla establece que si implementamos una interfaz, tenemos que implementar **todos** los métodos descritos en ella. De lo contrario se producirán errores de compilación.

Podemos describir la relación entre una clase y su interfaz mediante un diagrama de clases de UML. Vea el ejemplo de la figura 23.1. En un diagrama de clases de UML, una interfaz se muestra como rectángulo. El nombre de la interfaz va después de la palabra **<><Interface>>**. La relación **implements** se muestra como una flecha punteada.



**Figura 23.1** Una clase y su interfaz.

## PRÁCTICA DE AUTOEVALUACIÓN

- 23.2** Escriba métodos que implementen características adicionales de la interfaz **GloboInterfaz**, para acceder al valor de *y* y modificarlo.

También podemos usar las interfaces para describir una estructura de herencia. Por ejemplo, suponga que deseamos describir una interfaz para un tipo **GloboColoreado** que es una subclase de la interfaz **Globo** antes descrita. Podemos escribir:

```

public interface GloboColoreadoInterfaz
 extends GloboInterfaz {
 void setColor(Color color);
}

```

Esta interfaz hereda a la interfaz `GloboInterfaz` y establece que la interfaz `GloboColoreadoInterfaz` incluye un método adicional para establecer el color de un objeto. De manera similar podríamos describir toda una estructura tipo árbol de clases como interfaces, describiendo solamente su apariencia externa y sus relaciones subclase-superclase.

## PRÁCTICA DE AUTOEVALUACIÓN

- 23.3** Escriba el código adicional necesario para que una clase `GloboColoreado` implemente a la interfaz `GloboColoreadoInterfaz`.

En resumen, podemos utilizar interfaces para describir:

- Las clases en un programa.
- La estructura de herencia en un programa, las relaciones “es un”.

Lo que las interfaces **no pueden** describir es lo siguiente:

- Las implementaciones de los métodos (esto es para lo que se crearon).
- Qué clases utilizan otras clases, las relaciones “tiene un” (para esto se requiere alguna otra notación).

Para usar las interfaces en el diseño de un programa, debemos escribirlas antes de empezar a codificar las clases.

Las interfaces son especialmente útiles en programas de mediano y gran tamaño en los que se utilizan más de unas cuantas clases. En programas extensos que requieren equipos de programadores, su uso es casi imprescindible para facilitar la comunicación entre los miembros del equipo. Las interfaces también complementan a los diagramas de clases como una forma de documentación del diseño de un programa.

## ● Interfaces e interoperabilidad

Los aparatos electrodomésticos como tostadores y hornos eléctricos tienen un cable de alimentación con un enchufe en su extremo. El diseño del enchufe es estándar (en todo un país) y asegura que el aparato se pueda utilizar en cualquier parte (dentro de ese país). Por lo tanto, la adopción de una interfaz común asegura la interoperabilidad. En Java podemos utilizar las interfaces de una manera similar para asegurar que los objetos exhiban una interfaz común. Al trabajar con un objeto de este tipo en cualquier parte del programa podemos estar seguros de que soporta todos los métodos especificados por la descripción de la interfaz.

Como ejemplo, vamos a declarar una interfaz llamada `Visualizable`. Cualquier clase que cumpla con esta interfaz debe incluir un método llamado `mostrar` que muestre el objeto en pantalla. La declaración de la interfaz es:

```
public interface Visualizable {
 void mostrar(Graphics papel);
}
```

Ahora vamos a escribir una nueva clase llamada **Cuadrado**, la cual representa objetos gráficos cuadrados. En el encabezado de la clase indicamos que implementa a **visualizable**. Dentro del cuerpo de la clase incluimos el método **mostrar**:

```
import java.awt.Graphics;
public class Cuadrado implements Visualizable {
 private int x, y, tamaño;
 public void mostrar(Graphics papel) {
 papel.setColor(Color.black);
 papel.drawRectangle(x, y, tamaño, tamaño);
 }
 // otros métodos de la clase Cuadrado
}
```

Como se indica en el encabezado, esta clase (y cualquier objeto creado a partir de ella) se apega a la interfaz **Visualizable**. Esto significa que podemos trabajar con cualquier objeto de esta clase en un programa y cuando necesitemos mostrarlo en pantalla podremos usar su método **mostrar** sin ningún problema.

## PRÁCTICA DE AUTOEVALUACIÓN

- 23.4** Deseamos escribir una nueva clase **Círculo** que implemente a la interfaz **Visualizable**. Escriba el encabezado de la clase.

## ● Las interfaces y la biblioteca de Java

La mayoría de los programas en este libro utilizan interfaces para realizar el manejo de eventos. Observe el encabezado de casi cualquier programa. Dice:

```
public class Cualquiera extends JFrame implements ActionListener
```

Esto significa que el programa se apega a la interfaz **ActionListener** de la biblioteca. Implica que el programa debe proveer todos los métodos descritos en esa interfaz. En este caso sólo hay un método: el método **actionPerformed**.

Por lo general, un programa crea un objeto botón:

```
JButton botón = new JButton("opríma");
```

Después llama al método **addActionListener** del objeto **botón** para indicarle cuál objeto debe llamar cuando ocurra un evento:

```
botón.addActionListener(this);
```

Por último, cuando ocurre un evento se hace una llamada al método **actionPerformed** del programa.

Ahora suponga que proveemos el método `actionPerformed` pero omitimos `implements ActionListener`. ¿Qué ocurriría? La respuesta es que el programa no se compilaría correctamente. Ésta sería la instrucción incorrecta:

```
botón.addActionListener(this);
```

Ésta es una llamada al método `addActionListener` de la clase `JButton`. El encabezado del método `addActionListener` es el siguiente:

```
public void addActionListener(ActionListener objeto)
```

El cual declara su parámetro como de tipo `ActionListener`. Por ende, este método espera que su parámetro implemente a la interfaz `ActionListener`. Y si no es así, el compilador se queja.

El resultado final es que el programador se ve forzado a:

1. Declarar la clase de manera que implemente a una interfaz.
2. Proveer el método asociado dentro de la clase.

Con lo cual asegura que el programa se compile y después se ejecute en forma correcta.

## ● Interfaces múltiples

Así como una TV tiene interfaces para una fuente de energía y una fuente de señal, también en Java podemos especificar que una clase implemente a varias interfaces. De esta manera, a pesar de que una clase sólo puede heredar de otra clase, puede implementar cualquier número de interfaces.

Java es un lenguaje que provee herencia simple: una clase puede heredar (o ser la subclase) sólo de una clase. La estructura de la clase es un árbol, con la raíz en la parte superior, en donde una clase puede tener varias subclases pero sólo una superclase. La figura 23.2 muestra las clases ilustrativas `Globo` y `Juego` como subclases de la superclase `JFrame`. Cada clase aparece sólo dentro de un árbol individual y cada clase tiene sólo una superclase.

Algunas veces sería conveniente que una clase heredara de más de una superclase, como se describe en el siguiente encabezado de clase y se muestra en la figura 23.3:

```
public class Juego extends JFrame, Thread // error
```

Pero este encabezado está mal, ya que intenta extender dos clases. A esto se le denomina *herencia múltiple*. Algunos lenguajes como C++ permiten herencia múltiple pero Java no. La herencia múltiple

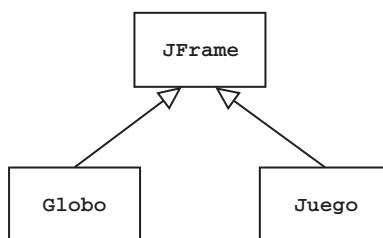
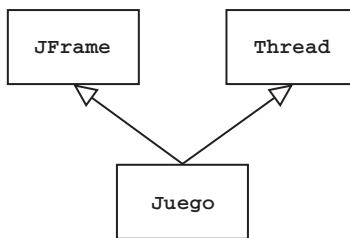


Figura 23.2 Herencia simple.



**Figura 23.3** Herencia múltiple (no tiene soporte en Java).

ple permite a una clase heredar conjuntos de métodos de varias clases, por lo que es potencialmente muy poderosa.

Si pensamos sobre los sistemas de clasificación en las ciencias y la naturaleza, a menudo se da el caso de que los objetos pertenecen a más de una clase. Por ejemplo, nosotros los humanos pertenecemos a una clase de género, pero también a una clase que prefiere cierto tipo de música. Entonces, todos pertenecemos a un árbol de herencia para los géneros, a otro para los gustos musicales, a otros para las lenguas maternas, y así por el estilo.

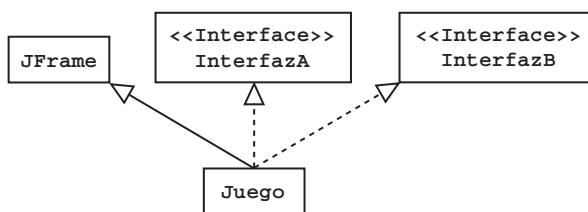
Java se diseñó para ser un lenguaje sencillo (pero poderoso) y, por lo tanto, permite sólo la herencia única en la que una clase sólo puede heredar de una superclase. Esto evita muchas de las confusiones de la herencia múltiple que no veremos aquí.

Sin embargo, hay una forma de emular una herramienta similar a la herencia múltiple en Java mediante el uso de interfaces. Esto se debe a que, mientras que una clase sólo puede extender a una clase individual, puede implementar a cualquier número de interfaces.

Las interfaces múltiples se muestran en la figura 23.4. Este ejemplo se codifica de la siguiente manera:

```
public class Juego extends JFrame implements InterfazA, InterfazB
```

Si **Juego** heredara de **InterfazA** y de **InterfazB**, heredaría un conjunto de métodos de ambas interfaces. Pero en vez de ello **Juego** implementa a las interfaces **InterfazA** e **InterfazB**, y estas interfaces no tienen métodos que ofrecer. Lo que esto significa es que la clase **Juego** está de acuerdo en proveer los métodos descritos en **InterfazA** e **InterfazB**; es decir, **Juego** está de acuerdo en apegarse a cierto comportamiento. El código para implementar **InterfazA** e **InterfazB** se tiene que escribir como parte de la clase **Juego**.



**Figura 23.4** Múltiples interfaces.

## Comparación entre interfaces y clases abstractas

Las interfaces y las clases abstractas son similares; la intención de esta sección es aclarar sus diferencias (en el capítulo 10 sobre herencia describimos las clases abstractas). El propósito de una clase abstracta es describir, como una superclase, las características comunes de un grupo de clases; se introduce mediante la palabra clave **abstract**. Las diferencias entre las clases abstractas y las interfaces son:

1. A menudo una clase abstracta provee una implementación de algunos de los métodos. En contraste, una interfaz nunca describe ninguna implementación.
2. Una clase puede implementar más de una interfaz, pero sólo puede heredar de una clase abstracta.
3. Una interfaz se utiliza en tiempo de compilación para realizar la comprobación. Por el contrario, una clase abstracta implica la herencia, lo cual conlleva vincular el método apropiado en tiempo de ejecución.
4. Una clase abstracta implica que las clases que la extiendan deben completar sus métodos **abstract**. Se espera la herencia. Pero una interfaz simplemente especifica el esqueleto para una clase, sin implicar que se utilizará para la herencia.

## Fundamentos de programación

Las interfaces completan el conjunto de herramientas proporcionadas por Java para describir las clases y sus interrelaciones. Las clases describen una colección de acciones y datos. Una interfaz describe la apariencia externa de una clase: los métodos que están públicamente disponibles. La herencia nos permite crear una nueva clase a partir de una clase anterior, con métodos adicionales útiles o métodos modificados. Una clase abstracta describe las características comunes de un grupo de clases, y puede proveer la implementación de algunos de los métodos compartidos.

## Errores comunes de programación

Recuerde que:

- Una clase sólo puede heredar de otra clase, incluyendo una clase abstracta.
- Una clase puede implementar cualquier cantidad de interfaces.

## Secretos de codificación

La descripción de una interfaz tiene la siguiente estructura:

```
public interface Nombre {
 void métodoA(parámetros);
 int métodoB(parámetros);
}
```

Los métodos son implícitamente `public`. Se omiten los cuerpos de los métodos.

## Nuevos elementos del lenguaje

- **interface** – la descripción de la interfaz externa para una clase (que tal vez no se haya escrito aún).
- **implements** – se utiliza en el encabezado de una clase para especificar que la clase implementa a una interfaz determinada.

## Resumen

- Las interfaces se utilizan para describir los servicios proporcionados por una clase.
- Las interfaces son útiles para describir la estructura de un programa. El compilador de Java puede verificar esta descripción.
- Las interfaces se pueden usar para asegurar que una clase se apega a una interfaz específica. Esto refuerza la interoperabilidad.
- Java soporta varias interfaces, pero sólo soporta la herencia simple.

## Ejercicios

### Interfaces como descripciones de diseño

- 23.1** Escriba una interfaz para describir métodos seleccionados de la clase `JTextField`.
- 23.2** Escriba una interfaz para describir una clase que represente cuentas bancarias. La clase se llama `Cuenta`. Tiene los métodos `depositar`, `retirar` y `getSaldoActual`. Decida los parámetros adecuados para los métodos.

- 23.3** Escriba interfaces para describir la estructura de un programa que consta de varias clases, como el programa del juego descrito en el capítulo 18 sobre diseño.

## Interfaces para la interoperabilidad

- 23.4** Escriba una clase llamada **Círculo** que describa objetos círculo y se apegue a la interfaz **Visualizable** que vimos antes.

### Respuestas a las prácticas de autoevaluación

```
23.1 int getY();
 void setY(int y);

23.2 public int getY() {
 return y;
 }

 public void setY(int nuevaY) {
 y = nuevaY;
 }

23.3 public class GloboColoreado implements GloboColoreadoInterfaz {

 private Color color;

 public setColor(Color nuevoColor) {
 color = nuevoColor;
 }
 }

23.4 public class Círculo implements Visualizable
```

# CAPÍTULO **24**



## Programación en gran escala: paquetes

En este capítulo conoceremos:

- Por qué se utilizan los paquetes.
- Cómo usar clases de un paquete mediante la instrucción `import`.
- Cómo agrupar clases dentro de un paquete mediante la instrucción `package`.
- Las reglas de alcance para los paquetes.

### ● Introducción

Las clases de las bibliotecas de Java se agrupan en paquetes. Todo programa de Java utiliza las clases de la biblioteca y, por lo tanto, utiliza paquetes. Algunos ejemplos de paquetes de la biblioteca de Java son `java.awt` y `javax.swing`.

Cuando escribimos un programa extenso con muchas clases, también es conveniente crear nuestros propios paquetes.

Las instrucciones y variables individuales se agrupan en un método. Después los métodos y las variables de instancia se agrupan en una clase. Finalmente, las clases se agrupan en un paquete. Los paquetes constituyen la estructura más grande en Java.

### ● Uso de clases y la instrucción `import`

Las bibliotecas de Java proveen miles de clases útiles. Por conveniencia, las clases se agrupan en paquetes. La mayoría de los programas de este libro empiezan con las siguientes instrucciones `import`:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
```

`java.awt`, `java.awt.event` y `javax.swing` son paquetes. Cada uno de estos paquetes contiene varias clases útiles. Por ejemplo, la clase `JButton` está en el paquete `javax.swing`. Las instrucciones `import` permiten que el programa utilice de manera conveniente las clases que están dentro de los paquetes. Como la instrucción `import` está presente, sólo necesitamos hacer referencia a `JButton` sin dificultades. Por ejemplo:

```
JButton botón = new JButton("iniciar");
```

Si se omitiera la instrucción `import` de todas formas podríamos usar la clase `JButton`, pero tendríamos que hacer referencia a ella con su nombre completo: `javax.swing.JButton`. Esto sería inconveniente e incómodo. Lo anterior nos demuestra el valor de la instrucción `import`.

Si únicamente necesitamos importar una sola clase de un paquete, por decir `JButton`, podemos escribirla:

```
import javax.swing.JButton;
```

Al usar `*` indicamos que queremos importar todas las clases del paquete especificado. De esta manera, si necesitamos varias clases de un paquete, es más sencillo utilizar la notación `*`.

Por conveniencia, Java importa dos paquetes de manera automática:

- El paquete `java.lang`.
- El paquete actual.

El paquete `java.lang` provee varias clases de uso común, incluyendo `String`, `Thread`, `Math`, `Integer` y `Double`.

## PRÁCTICA DE AUTOEVALUACIÓN

- 24.1** Las clases llamadas `Lunes`, `Martes`, `Miércoles`, `Jueves`, `Viernes`, `Sábado` y `Domingo` están agrupadas en un paquete llamado `semana`. Escriba la instrucción `import` que se necesitará para usar la clase `Viernes`. Escriba la instrucción para crear un objeto `viernes` de la clase `Viernes`. Escriba la instrucción `import` que se necesitará para usar todas las clases del paquete.

### ● Creación de paquetes mediante la instrucción `package`

En este libro, el programa más complejo sólo tiene cinco clases. Pero un programa extenso puede consistir de decenas o cientos de clases. Con un programa extenso a menudo es conveniente agrupar las clases en paquetes y que el programador invente un nombre para cada paquete. Las razones de usar paquetes son:

- Es una buena forma de controlar la complejidad de un programa extenso.
- Las clases de un mismo paquete tienen derechos de acceso especiales entre sí.

Así, el programador agrupa en el mismo paquete las clases que son similares o se relacionan de manera estrecha entre sí.

Suponga que un programa consta de tres grupos de clases:

- Clases que manejan la interfaz de usuario.
- Clases que acceden a la base de datos.
- Clases que manejan la lógica central del programa.

Creamos tres paquetes llamados **gui**, **basedatos** y **lógica**. Observe que los nombres de los paquetes empiezan con letra minúscula. A continuación necesitamos asegurar que las clases individuales estén en el paquete correcto. Esto se logra mediante la instrucción **package**, que se escribe en el encabezado de la clase. Por ejemplo, si tenemos una clase llamada **InicioSesión** que maneja la parte de la GUI relacionada con el inicio de sesión, escribimos lo siguiente en el encabezado de la clase:

```
package gui;
public class InicioSesión
```

Los programas que presentamos en este libro no tienen una instrucción **package** en su encabezado. Si omite una instrucción **package** significa que la clase se coloca, de manera predeterminada, en un paquete sin nombre.

## PRÁCTICA DE AUTOEVALUACIÓN

- 24.2** Una clase llamada **Respaldo** se debe colocar en un paquete llamado **basedatos**. Escriba la instrucción **package** y el encabezado de la clase.

## ● Paquetes, archivos y carpetas

Ya vimos que es una buena práctica:

- Colocar cada clase en su propio archivo (con la extensión **.java**).
- Asignar al nombre de archivo el mismo nombre que la clase.

Al crear un paquete:

- Cree una carpeta con el mismo nombre que el paquete.
- Coloque todos los archivos de clases para ese paquete en la carpeta.
- Coloque la carpeta del paquete en la misma carpeta que el archivo que contiene el código que importa el paquete.

Esto asegura que el compilador y el sistema en tiempo de ejecución puedan encontrar todos los archivos.

## PRÁCTICA DE AUTOEVALUACIÓN

- 24.3 Las clases llamadas **Lunes**, **Martes**, **Miércoles**, **Jueves**, **Viernes**, **Sábado** y **Domingo** están agrupadas en un paquete llamado **semana**. Mencione los archivos y el directorio (carpeta) que se necesitan. Escriba las instrucciones **package** necesarias.

## Reglas de alcance

Cuando creamos y usamos paquetes entran en juego algunas nuevas reglas de alcance. La esencia es que las clases del mismo paquete acceden unas a otras con mucha facilidad.

Al escribir un método especificamos que es **private**, **public**, **protected** o simplemente no le asignamos un prefijo. El prefijo determina quién puede acceder al método. Como hemos visto, **public** significa que el método está disponible para todos. **private** significa que sólo se puede acceder a él desde el interior de la clase. **protected** significa que el método puede ser utilizado por esa clase, por cualquier subclase de esa clase y por cualquier otra clase del mismo paquete.

Si no asignamos un prefijo a un método, significa que se puede acceder a ese método desde cualquier parte dentro del mismo paquete, pero es inaccesible desde el exterior. Esto también se aplica a las clases, los constructores y las variables. Esto significa que el programador puede establecer una relación estrecha entre los métodos en el mismo paquete y va de acuerdo con la idea de que las clases del mismo paquete están relacionadas entre sí.

En el apéndice G se proporcionan las reglas completas de visibilidad.

## Los paquetes de la biblioteca de Java

He aquí algunos de los paquetes de Java con una descripción general de su contenido:

|                    |                                                                                                                                                                      |
|--------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>java.lang</b>   | Contiene las clases que soportan las principales características del lenguaje, como <b>Object</b> , <b>String</b> , número, excepción e hilos.                       |
| <b>java.util</b>   | Son clases utilitarias convenientes, como <b>Random</b> y <b>ArrayList</b> .                                                                                         |
| <b>java.io</b>     | Flujos de texto de entrada y salida para caracteres y números.                                                                                                       |
| <b>java.net</b>    | Clases que llevan a cabo funciones de red, programación de sockets, interacción con Internet.                                                                        |
| <b>javax.swing</b> | Aquí se incluyen las clases para proveer componentes de GUI como botones ( <b>JButton</b> ), etiquetas ( <b>JLabel</b> ) y controles deslizables ( <b>JSlider</b> ). |
| <b>java.awt</b>    | awt significa Kit de herramientas de Ventanas Abstractas. Aquí están los métodos gráficos como <b>drawLine</b> .                                                     |
| <b>java.applet</b> | Las clases proporcionan soporte para los applets de Java (programas que se ejecutan desde un navegador Web).                                                         |

## ● Errores comunes de programación

Es muy tentador usar una instrucción `import` para facilitar el uso de ciertos métodos y constantes de clases de las bibliotecas. Por ejemplo, en vez de decir `Math.sqrt` y `Color.white` parecería más fácil incluir instrucciones `import`:

```
import java.lang.Math.*;
import java.awt.Color.*;
```

y después simplemente hacer referencia a `sqrt` y `white`. Sin embargo, esto no funciona. Una razón es que la gramática de Java exige que al método de una clase siempre se le anteponga el nombre de su clase. La otra razón es que la instrucción `import` nos permite evitar el uso del nombre de un paquete, no el nombre de una clase.

## Nuevos elementos del lenguaje

- `import` – permite al usuario un fácil acceso a las clases en un paquete.
- `package` – especifica que la clase se debe colocar en el paquete con el nombre especificado.

## Resumen

Los paquetes son una forma de agrupar clases. Las diversas clases de la biblioteca de Java están agrupadas en paquetes convenientes.

Para usar las clases de un paquete tenemos que proporcionar el nombre completo del paquete y la clase. Una alternativa más conveniente es el uso de instrucciones `import`.

A menudo es conveniente agrupar las clases de un programa extenso en distintos paquetes. A cada paquete se le asigna un nombre. Las clases se pueden colocar en el paquete apropiado, para lo cual empleamos la instrucción `package`.

Las reglas de alcance de Java implican que las clases dentro del mismo paquete tienen derechos de acceso especial entre sí.

## ● Ejercicio

- 24.1** Es poco probable que usted vaya a crear sus propios paquetes, a menos que escriba programas que sean más grandes que los que vimos o sugerimos como ejercicios en este libro. Sin embargo, las clases de biblioteca están agrupadas en paquetes. Si tiene un navegador, explore las bibliotecas de Java. Empiece con el paquete `java.util`. De manera alternativa puede consultar el apéndice A para ver cómo se organizan las clases de biblioteca en paquetes.

## Respuestas a las prácticas de autoevaluación

- 24.1** Para usar la clase **Viernes**, escriba:

```
import semana.Viernes;
```

Para crear un objeto de la clase **Viernes**, escriba:

```
Viernes viernes = new Viernes();
```

Para usar todas las clases del paquete escriba:

```
import semana.*;
```

- 24.2**
- ```
package basedatos;  
public class Respaldo
```

- 24.3** Los archivos son **Lunes.java**, **Martes.java**, etc.

Hay que nombrar **semana** a la carpeta.

La primera línea de cada archivo debería ser:

```
package semana;
```

CAPÍTULO 25



Polimorfismo

En este capítulo conoceremos:

- Cómo usar el polimorfismo.
- Cuándo utilizar el polimorfismo.

● Introducción

Le presentaremos la idea del polimorfismo con un ejemplo sencillo. Suponga que tenemos dos clases llamadas **Esfera** y **Burbuja**. Podemos crear una instancia de **Esfera** y una instancia de **Burbuja** de la forma usual:

```
Esfera esfera = new Esfera();
Burbuja burbuja = new Burbuja();
```

Suponga que cada clase tiene un método llamado **mostrar**. Entonces podemos mostrar en pantalla los dos objetos de la siguiente manera:

```
esfera.mostrar(papel);
burbuja.mostrar(papel);
```

y aunque estas dos llamadas son muy similares, en cada caso llamamos a la versión apropiada de **mostrar**. Aunque hay dos métodos con el mismo nombre (**mostrar**), son distintos y el sistema de Java se asegura de que siempre se seleccione el método correcto. De esta forma, cuando llamamos al método **mostrar** para el objeto **esfera**, se hace una llamada al método definido dentro de la clase **Esfera**. Y cuando llamamos a **mostrar** para el objeto **burbuja**, se hace una llamada al método definido dentro de la clase **Burbuja**. Ésta es la esencia del polimorfismo.

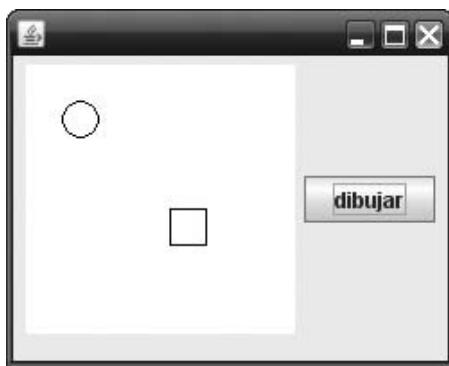


Figura 25.1 Despliegue de las figuras utilizando polimorfismo.

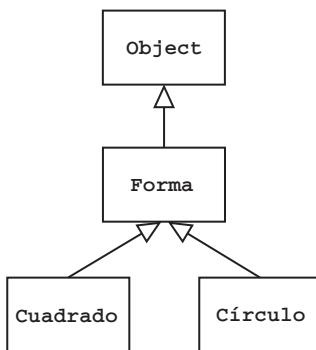
● El polimorfismo en acción

En este capítulo utilizaremos como ejemplo un programa que muestra formas gráficas en la pantalla: cuadrados, círculos y similares (figura 25.1). El programa utiliza una clase abstracta llamada **Forma**, la cual describe todos los atributos compartidos de estas formas, incluyendo su ubicación en la pantalla (en el capítulo 10 sobre la herencia explicamos las clases abstractas). He aquí la clase **Forma**:

```
import java.awt.Graphics;
public abstract class Forma {
    protected int x, y;
    protected int tamaño = 20;
    public abstract void mostrar(Graphics áreaDibujo);
}
```

Cada forma se describe mediante su propia clase, una subclase de la clase **Forma** (figura 25.2). Por ejemplo, las clases **Círculo** y **Cuadrado** son:

```
import java.awt.Graphics;
public class Círculo extends Forma {
    public Círculo(int xInic, int yInic) {
        x = xInic;
        y = yInic;
    }
}
```

**Figura 25.2** Diagrama de clases para las clases de formas.

```

public void mostrar(Graphics áreaDibujo) {
    áreaDibujo.drawOval(x, y, tamaño, tamaño);
}
}

import java.awt.Graphics;
public class Cuadrado extends Forma {
    public Cuadrado(int xInic, int yInic) {
        x = xInic;
        y = yInic;
    }

    public void mostrar(Graphics áreaDibujo) {
        áreaDibujo.drawRect(x, y, tamaño, tamaño);
    }
}
  
```

He aquí un programa que utiliza estas clases para crear dos formas. Al hacer clic en el botón, las formas se almacenan en un objeto `ArrayList` llamado `grupo` y se muestran en pantalla (un objeto `ArrayList`, que vimos en el capítulo 12, es una estructura de datos conveniente que se expande o contrae para alojar los datos requeridos). La salida se muestra en la figura 25.1.

```

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.util.*;

public class Formas extends JFrame implements ActionListener {
    private JButton botón;
    private JPanel panel;
  
```

```

public static void main(String[] args) {
    Formas demo = new Formas();
    demo.setSize(250,200);
    demo.crearGUI();
    demo.setVisible(true);
}

private void crearGUI() {
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    Container ventana = getContentPane();
    ventana.setLayout(new FlowLayout());

    panel = new JPanel();
    panel.setPreferredSize(new Dimension(150, 150));
    panel.setBackground(Color.white);
    ventana.add(panel);

    botón = new JButton("dibujar");
    ventana.add(botón);
    botón.addActionListener(this);
}

public void actionPerformed(ActionEvent event) {
    Graphics papel = panel.getGraphics();

    Círculo círculo = new Círculo(20, 20);
    Cuadrado cuadrado = new Cuadrado(80, 80);
    ArrayList <Forma> grupo = new ArrayList <Forma>();
    grupo.add(círculo);
    grupo.add(cuadrado);

    papel.setColor(Color.white);
    papel.fillRect(0, 0, 150, 150);
    papel.setColor(Color.black);
    for (Forma forma : grupo) {
        forma.mostrar(papel);
    }
}
}

```



En este programa se utiliza el polimorfismo: se hace una llamada al método `mostrar` en dos ocasiones con distintos resultados, de acuerdo con el objeto que se esté usando. Podemos ver que las dos llamadas de `mostrar` dentro del ciclo `for`:

```
forma.mostrar(papel);
```

proporcionan dos resultados distintos. Esto no es necesariamente lo que podríamos esperar, pero es totalmente correcto. Se muestran dos resultados distintos debido a que el sistema de Java selecciona de manera automática la versión de `mostrar` asociada con la clase del objeto. Cuando se llama por primera vez al método `mostrar`, la variable `forma` contiene el objeto `círculo` y por lo tanto

se hace una llamada a la versión `mostrar` de la clase `Círculo`. Después ocurre lo mismo con `cuadrado`.

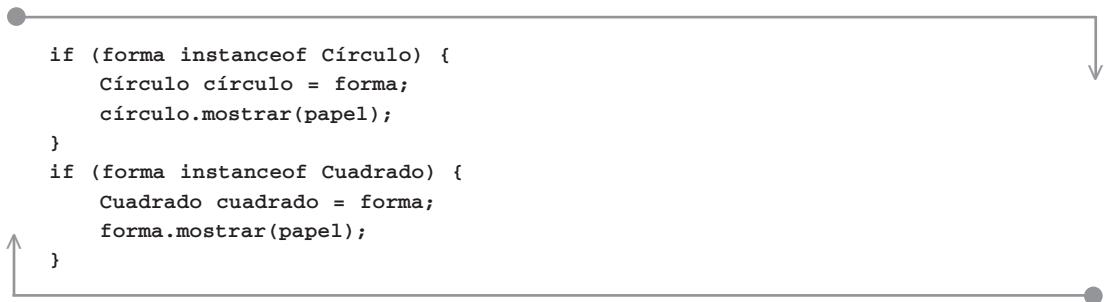
La clase de un objeto se determina al momento de crearlo mediante `new` y sigue siendo la misma sin importar lo que ocurra con el objeto. No importa qué haga usted con un objeto en un programa, siempre retendrá las características que tenía al momento en que se creó. En la analogía de la familia, usted retiene su identidad y su relación con sus ancestros aunque se case, cambie su nombre o cambie de país. Sin duda esto tiene sentido. El eslogan apropiado podría ser: “un cuadrado siempre será un cuadrado”. En nuestro programa, aun cuando `cuadrado` se almacena en un objeto `ArrayList` de tipo `Forma`, sigue siendo un `Cuadrado`.

Al llamar a un método, el polimorfismo se asegura de seleccionar la versión apropiada de ese método. La mayor parte del tiempo, cuando programamos en Java no estamos conscientes de que el sistema de Java está seleccionando el método correcto para la invocación. Es un proceso automático e invisible.

El polimorfismo nos permite escribir una sola y poderosa instrucción tal como:

```
forma.mostrar(papel);
```

En vez de una serie de instrucciones `if` como se muestra a continuación:



```
if (forma instanceof Círculo) {
    Círculo círculo = forma;
    círculo.mostrar(papel);
}
if (forma instanceof Cuadrado) {
    Cuadrado cuadrado = forma;
    forma.mostrar(papel);
}
```

Sin duda este código es burdo y extenso. Utiliza la palabra clave `instanceof` para preguntar si un objeto es miembro de una clase con nombre (sí, así se escribe). Si hubiera una gran cantidad de formas, habría una cantidad igual de instrucciones `if`. Al evitar esto se demuestra qué tan poderoso y conciso es el polimorfismo.

Como hemos visto en este pequeño ejemplo, el polimorfismo comúnmente ayuda a reducir un segmento de un programa y lo hace más ordenado por medio de la eliminación de una serie de instrucciones `if`. Pero este logro es más considerable de lo que parece. Significa que instrucciones tales como:

```
forma.mostrar(papel);
```

no saben nada acerca de la posible variedad de objetos que se puedan utilizar como el valor de `forma`. Por lo tanto, se extiende el ocultamiento de información (que ya está presente en gran medida en un programa OO). Para verificar esto podemos valorar qué tanto tendríamos que modificar este programa para adaptar un nuevo tipo de forma (alguna subclase adicional de `Forma`), como una elipse. La respuesta es que no tendríamos que modificarlo para nada. Esto significa que el programa es mucho muy flexible. Por ende, el polimorfismo mejora la modularidad, la capacidad de reutilización y de mantenimiento.

Fundamentos de programación

El polimorfismo representa el tercer elemento más importante de la POO. El conjunto completo de los tres elementos es:

1. Encapsulamiento: significa que los objetos se pueden hacer altamente modulares.
2. Herencia: significa que las características deseables en una clase existente se pueden reutilizar en otras clases sin afectar la integridad de la clase original.
3. Polimorfismo: significa diseñar código que pueda manipular con facilidad objetos de distintas clases. Las diferencias entre los objetos similares se pueden procesar sin problemas.

Por lo general, los programadores principiantes empiezan utilizando el encapsulamiento, más tarde pasan a la herencia y después utilizan el polimorfismo.

El caso de estudio que vimos en este capítulo utiliza una diversidad de objetos (formas) con factores comunes incorporados en una superclase. Ahora sabemos que la facilidad de la herencia ayuda a describir la similitud de grupos de objetos de una forma económica. El otro lado de la moneda es el uso de objetos, y aquí es en donde el polimorfismo nos ayuda a usarlos de una manera concisa y uniforme. La diversidad no se manipula mediante una proliferación de instrucciones `if`, sino mediante una sola llamada a un método. Por ende, si llega a ver código como el siguiente:

```
if (forma instanceof Círculo) {  
    forma.mostrarCírculo(papel);  
}  
if (forma instanceof Cuadrado) {  
    forma.mostrarCuadrado(papel);  
}
```

debe pensar mejor en usar el polimorfismo. Como `mostrarCírculo` y `mostrarCuadrado` representan un concepto similar, podemos hacer este concepto un método de una superclase y rediseñar la secuencia de este código de una manera más simple:

```
forma.mostrar(papel);
```

Así estamos usando el polimorfismo para seleccionar el método apropiado. Cuando esto ocurre, se selecciona la versión del método `mostrar` que coincide con el objeto actual. Esto sólo se puede decidir cuando el programa está en ejecución, justo antes de llamar al método. A esto se le conoce como *vinculación tardía*, *vinculación dinámica* o *vinculación postergada*. Es una característica esencial de un lenguaje con soporte para el polimorfismo.

En resumen, un objeto (una vez creado) retiene su identidad fundamental, sin importar lo que hagamos con él. Una analogía: aun cuando usted cambie su nombre, sigue siendo la misma persona; sus genes no cambian. Si coloca un objeto en una variable de un tipo distinto (como un objeto `ArrayList`), el objeto sólo se puede asignar a una variable cuya clase sea una superclase del objeto o una subclase del objeto. En otras palabras, las cosas se mantienen estrictamente dentro de la familia.





Fundamentos de programación (continúa)

El polimorfismo ayuda a construir programas que sean:

- Concisos (más cortos de lo que podrían ser si no se utilizara).
- Modulares (las partes no relacionadas se mantienen separadas).
- Fáciles de modificar y adaptar (por ejemplo, cuando se introducen nuevos objetos).

En general, la metodología para explotar el polimorfismo dentro de un programa específico es la siguiente:

1. Identificar las similitudes (métodos y variables comunes) entre los objetos o clases en el programa.
2. Diseñar una superclase que abarque las características comunes de las clases.
3. Diseñar las subclases que describan las características únicas de cada una de las clases, al tiempo que hereden las características comunes de la superclase.
4. Identificar cualquier lugar en el programa en donde se deba aplicar la misma operación a cualquiera de los objetos similares. Tal vez se vea tentado a usar instrucciones `if` en esta ubicación. Por el contrario, éste es el lugar adecuado para usar el polimorfismo.
5. Asegurarse de que la superclase contenga un método abstracto correspondiente al método que se va a utilizar de manera polimórfica.

Errores comunes de programación

Si piensa explotar el polimorfismo para agrupar varias clases bajo una sola superclase, debe asegurarse de que la superclase describa a todos los métodos que se utilizarán en cualquier instancia de la superclase. Algunas veces para ello se requerirán métodos abstractos en la superclase cuyo único propósito sea permitir que el programa se compile.

Nuevos elementos del lenguaje

- La palabra clave `instanceof` permite a un programa evaluar un objeto para ver a qué clase pertenece.

Resumen

Los principios del polimorfismo son:

1. Un objeto siempre retiene la identidad de la clase a partir de la cual se creó (un objeto nunca se podrá convertir en un objeto de otra clase).
2. Cuando se utiliza un método en un objeto, siempre se utiliza el método asociado con la clase del objeto.

El polimorfismo permite ocultar información y reutilizar código, ya que nos ayuda a que las piezas de código se puedan aplicar en muchas áreas.

Ejercicios

- 25.1 Una clase abstracta llamada **Animal** tiene un método constructor, los métodos **setPeso** y **getPeso**, además de un método de función **dice**. El valor de **peso** representa el peso del animal, un valor **int**. El método **dice** devuelve una cadena, la cual representa el ruido que hace el animal.

La clase **Animal** es:

```
abstract class Animal {  
    protected int peso;  
    public Animal() {}  
    public void setPeso(int peso) {  
        this.peso = peso;  
    }  
    public int getPeso() {  
        return peso;  
    }  
    public abstract String dice();  
}
```

La clase **Animal** tiene las subclases **Vaca**, **Serpiente** y **Cerdo**. Estas subclases tienen distintas implementaciones de **dice**, las cuales devuelven los valores “mu”, “sss” y “oinc”, respectivamente. Escriba las tres subclases. Cree los objetos **daisy**, **sid** y **wilbur**, respectivamente, a partir de las tres clases, agréguelas a un objeto **ArrayList** y utilícelos sus métodos. Muestre los sonidos que hacen los animales en un área de texto.

25.2 En el programa de las formas que vimos en el libro, agregue una nueva forma (una línea recta) a la colección de formas disponibles. Use el método de biblioteca `drawLine` para dibujar un objeto línea. Agregue código para crear un objeto línea, agregarlo a la lista de objetos (en el objeto `ArrayList`) y mostrarlo en pantalla junto con las otras formas.

25.3 Mejore el programa formas para convertirlo en un paquete completo de dibujo que permita seleccionar formas de un menú y colocarlas en la ubicación deseada en un panel. El usuario debe especificar la posición con un clic del ratón. Así, el usuario:

1. Primero hace clic en alguna parte del panel.
2. Despues hace clic en el elemento deseado del menú.

Debe proveer también un botón “borrar”.

25.4 Un banco ofrece a sus clientes dos tipos de cuentas: una cuenta regular y una cuenta dorada. Los dos tipos de cuentas proveen varias características compartidas pero también ofrecen otras características únicas. Las características comunes son:

- Abrir una cuenta con un nombre y un saldo inicial.
- Mantener un registro del saldo actual.
- Depositar o retirar una cantidad de dinero.

Cada vez que se hace un retiro en una cuenta regular, hay que verificar que no se sobregire. El dueñahabiente de una cuenta dorada puede sobregirarse de manera indefinida. En una cuenta regular el interés se calcula a razón del 5% anual de la cuenta. Una cuenta dorada tiene una tasa de interés del 6% menos un cargo fijo de \$100 al año.

Escriba una clase que describa las características comunes y clases individuales para describir las cuentas regular y dorada.

Construya un programa para utilizar estas clases. Cree dos cuentas bancarias: una regular y otra dorada. Cada cuenta se debe crear con el nombre de una persona y cierta cantidad inicial de dinero. Coloque los objetos que representan cada cuenta en un objeto `ArrayList`. Muestre el nombre, saldo e interés de las cuentas en un área de texto al hacer clic en un botón (suponga por cuestión de simplicidad que las cantidades de dinero se mantienen como valores `int`).

CAPÍTULO **26**



Java en contexto

En este capítulo conoceremos:

- Los principios de diseño detrás de Java.
- Microsoft y Java.
- Java y la internacionalización.
- Java y las bases de datos.
- Java e Internet: RMI, JSP, servlets, beans.
- Java y JavaScript.

● Introducción

En este capítulo explicaremos cómo se relaciona Java en las áreas de la computación e Internet. Además, Java se actualiza de manera constante a través de nuevas bibliotecas de clases, herramientas y tecnologías. Aquí le proporcionaremos las generalidades del ámbito actual. Empezaremos por revisar las principales características de Java que se resumen en un documento conocido como “White Paper” de Sun MicroSystems, empresa que lanzó el lenguaje. Las características del lenguaje Java son:

- Simple.
- Orientado a objetos.
- Listo para trabajar en red.
- Interpretado.
- Robusto.
- Seguro.
- De arquitectura neutral.

- Portable.
- De alto rendimiento.
- Multihilos.
- Dinámico.

Simple

Java surgió en parte como reacción a una complejidad percibida en otros lenguajes importantes (principalmente C++). Algunos consideran a C++ como un lenguaje extenso, difícil de manejar y de aprender, que muestra conceptos inconsistentes y puede darse el caso de que un usuario lo utilice mal por accidente. En contraste, Java es pequeño y bello. Por ejemplo, mientras que C++ tiene herencia múltiple, Java tiene la herencia simple, que es más sencilla. En C++ el programador puede manipular directamente los apuntadores a objetos en memoria, pero esta práctica altamente peligrosa no está disponible para el programador de Java.

Claro que debemos diferenciar el lenguaje de sus bibliotecas. Ahora hay cientos de clases y miles de métodos; se requiere tiempo para dominar algo así. Pero el lenguaje es relativamente simple.

Simpleza significa facilidad de aprender, que los usuarios dominen el lenguaje y que haya menos errores. Java se diseñó como un lenguaje pequeño, con la esperanza de que lo comprendieran por completo, implementaran y adoptaran muchas personas. A medida que han evolucionado, la mayoría de los lenguajes se han vuelto cada vez más grandes, y tal vez ocurra lo mismo con Java. Como los diseñadores de Java dicen: “teniendo en cuenta que el sistema crecerá cada vez más en las siguientes versiones, optamos por empezar con lo mínimo”. Esperemos que prevalezca su visión de diseño original.

Orientado a objetos

Java es un lenguaje orientado a objetos (OO). En este libro explicamos que esto significa que un programa se construye a partir de objetos, los que a su vez se crean a partir de clases. Esto promueve la modularidad de los programas. Las clases pueden heredar variables y métodos de las superclases. Esto es muy poderoso como soporte para la reutilización de software. En consecuencia, los programas OO son por lo general más pequeños que los programas equivalentes que no son OO. Otra ventaja es el aumento en la confiabilidad que se obtiene al usar software comprobado en vez de empezar desde cero.

En la actualidad el paradigma de la OO domina la computación, no sólo en la programación sino también en el análisis y diseño.

Independencia de la plataforma (portabilidad)

¿Qué tanto trabajo se requiere para mover (portar) un programa a una plataforma distinta? En esencia, ésta es la cuestión de la portabilidad. Java se diseñó como un lenguaje que se puede portar con facilidad a cualquier computadora y sistema operativo. En la actualidad se ejecuta en todas las principales computadoras y sistemas operativos. Para lograr esta portabilidad, Java se compila en código independiente de la máquina, al cual se le conoce como código de bytes. Éste es código para una

computadora “ficticia”: una que se pueda simular en software. Al software de simulación se le denomina Máquina Virtual de Java (JVM). Es necesaria una JVM para cada tipo de computadora que ejecuta Java; se ha escrito una JVM para casi todas las computadoras. De hecho, una JVM es un programa comparativamente pequeño (aproximadamente 256 Kbytes) y es bastante fácil de escribir. La JVM está disponible para ejecutarse en muchos tipos distintos de procesadores, no sólo de Intel. La JVM tampoco está atada a un sistema operativo específico, como Windows, Unix o GNU/Linux. Por ende, un programa de Java se puede distribuir (como código de bytes) y ejecutar en una gran variedad de máquinas.

Al compilar un programa de Java, la información sobre los métodos y variables de biblioteca utilizados se retiene en el archivo de código de bytes. Generalmente, la siguiente fase es la vinculación, en donde se combina el código de bytes del programa con el código de bytes de las bibliotecas que se hayan utilizado. En Java se pospone la vinculación con las bibliotecas hasta que el programa se ejecute. Esta vinculación dinámica tiene muchas ventajas. En primer lugar, significa que el código de bytes de Java se puede transmitir con más rapidez alrededor de la red, sin el impedimento del software de biblioteca. Esto aumenta la velocidad. En segundo lugar, cuando el código de Java llega a un sitio y empieza su ejecución, se vincula con la biblioteca de Java específica para ese sitio. Esto mejora la portabilidad. Por último, cuando se actualiza cualquiera de las bibliotecas, el programa de Java selecciona la versión más reciente de manera automática. Esto ayuda mucho a mantener y actualizar el software.

Otra característica de Java que contribuye a la portabilidad es el hecho de que los tipos de datos se definen de manera que tengan tamaños que sean independientes de una máquina individual. Por ejemplo, un entero siempre es de 32 bits, sin importar la máquina. Esto significa que el programador puede estar seguro del comportamiento de un programa, en donde sea que se ejecute. Esta estandarización tan sencilla no está presente en otros lenguajes, en especial en C y C++, en donde el tamaño de los tipos de datos cambia de una máquina a otra.

La JVM no está atada de manera exclusiva a la ejecución de Java. También provee una plataforma portable lista para usarse para los creadores de lenguajes, quienes pueden crear compiladores que generen código de bytes. Los lenguajes notables producidos de esta forma son Groovy, un lenguaje de secuencias de comandos al estilo Ruby/Python, y Scala, que es un lenguaje de propósito general cuya popularidad va en aumento.

Rendimiento

La transmisión de programas de Java por Internet o dentro de una intranet es rápida, ya que el código de Java es compacto. Además, el software de biblioteca no se transmite con el programa, sino que se vincula en tiempo de ejecución en el equipo de destino. Cualquier problema de rendimiento con Java surge en tiempo de ejecución, aunque las recientes mejoras a la JVM hacen que Java se acerque a la velocidad de C++.

No obstante, debe tener cuidado al evaluar las pruebas de velocidad de los lenguajes. Algunos de los ejemplos que se utilizan son programas pequeños orientados a números, por lo que no podríamos usar dichos resultados para predecir (por ejemplo) el tiempo de respuesta de una GUI.

Otro factor de rendimiento es la velocidad de la computadora en la que se ejecuta el programa. No siempre es conveniente la máxima velocidad; algunas veces basta tener velocidad “suficiente”. Además, la velocidad de las computadoras aumenta año con año. Desde luego que Java nos brinda la portabilidad, algo que los lenguajes más rápidos no proveen en muchos casos.

Seguridad

Hay dos estilos de programas de Java: aplicaciones (que vimos durante todo el libro) y applets, que son programas que se descargan de un servidor por medio de una página Web. Se ejecutan en la computadora del usuario.

Los applets (que se pueden ejecutar sin que nosotros sepamos) imponen problemas de seguridad. Un programa desconocido podría:

- Hacer que falle su equipo.
- Eliminar archivos.
- Usar y transmitir información privada a otro equipo.
- Llenar su espacio de disco.
- Introducir un virus.

Dichos daños podrían ser deliberados o accidentales; el resultado es el mismo. Las preocupaciones sobre esos daños restringen el uso de Internet y los usuarios necesitan protegerse. Java provee los mecanismos necesarios para que los usuarios se protejan a sí mismos de los programas malintencionados. Sin duda éste fue uno de los principales objetivos de diseño de Java. Para ello utiliza toda una serie de dispositivos que describiremos a continuación.

El lenguaje Java en sí

A diferencia de otros lenguajes, Java no permite el uso de apuntadores. En algunos lenguajes se permite al programador establecer apuntadores, por lo general a estructuras de datos. Pero una vez que tiene el acceso a un apuntador, el programador puede entonces acceder, ya sea por accidente o por diseño, a cualquier cosa en memoria. Por ejemplo (si se permitiera), un programa podría acceder a los datos privados dentro de un objeto. Así, la prohibición de apuntadores en Java es una medida de seguridad en sí misma.

Java es un lenguaje fuertemente tipificado. Esto significa que si, por ejemplo, declaramos una variable como entera, el compilador se asegura de que se utilice de manera consistente como entera y no como carácter, por ejemplo. Si el programador desea convertir una pieza de datos de un tipo a otro, tiene que hacerlo de manera explícita (a esto se le conoce como *conversión de tipos*). La verificación en tiempo de compilación de este tipo es económica y efectiva en comparación con el proceso de detectar y erradicar los errores que se hacen evidentes sólo cuando el programa está en ejecución.

El lenguaje Java es totalmente orientado a objetos; un aspecto principal de esto es el encapsulamiento. Las variables y los métodos dentro de un objeto que se declaran como privados no se pueden utilizar desde el exterior del objeto. De esta forma hay un control claro y explícito sobre el acceso a los datos y las acciones.

El lenguaje Java evita que un programador acceda a una variable que no se ha inicializado. Algunos otros lenguajes permiten acceso a una variable sin inicializar. Éste es un error del programador al suponer que lo hicieron de manera explícita o que el sistema lo hace por él de manera implícita.

El verificador

Cuando llega una nueva pieza de código de bytes de Java para ejecutarse, el verificador se asegura de que sea segura. Esto es para minimizar cualquier riesgo de que el código se haya dañado en el

camino, o que se haya creado código inseguro mediante una herramienta distinta al compilador regular de Java. Las verificaciones aseguran que:

- El código no cree apuntadores peligrosos.
- El acceso a las variables y los métodos privados sea válido.
- Los métodos se llamen con parámetros del tipo correcto.
- No se realicen conversiones de datos ilegales.

El cargador de clases

Una de las maneras en que un villano podría penetrar las defensas de Java es al proporcionar algunas clases que simulen ser clases estándar de Java. Si dichas clases se cargaran por error en vez de las clases apropiadas, se podría provocar cualquier tipo de daños. Para evitar esto, el cargador de clases distingue con cuidado entre los distintos paquetes y en especial entre las clases locales integradas y cualquier otra clase.

La Máquina Virtual de Java (JVM)

Una vez verificado, el código de bytes no se ejecuta directamente en la máquina, sino que se interpreta. Esto significa que el programa no puede acceder a las instrucciones puras de máquina (con todo el poder y peligro que eso implicaría). Por medio de este mecanismo se niega toda una serie de facilidades al programa:

- Acceso a las áreas de memoria que contienen información del sistema operativo.
- Acceso a los archivos y a Internet por medio de applets.

Así, un programa de Java se ejecuta en una caja o jaula (algunas veces conocida como caja de arena) y todo lo que hace es restringido.

Arreglos

Una forma potencial en la que un programa malintencionado podría hacer daño es al declarar un arreglo y después acceder a elementos inexistentes: elementos con un subíndice mayor que el tamaño del arreglo. Esto se evita en Java debido a que la JVM comprueba que un subíndice esté dentro de los límites definidos.

Recolección de basura

Cuando ya no se utiliza un objeto en un programa de Java, el sistema libera de manera automática la memoria que se le asignó. A esto se le conoce como *recolección automática de basura*. En lenguajes como C++, es responsabilidad del programador invocar de manera explícita al sistema operativo para informarle que una pieza de memoria está libre. Los siguientes errores pueden ocurrir (y ocurren) en dichos lenguajes:

- No liberar memoria cuando ya no se necesita, lo cual provoca que el programa acumule regiones cada vez mayores de memoria.

- Liberar la misma pieza de memoria dos veces, provocando ligeros errores de corrupción de memoria que pueden ser difíciles de localizar.
- Liberar memoria pero seguir utilizándola, provocando problemas similares.

Estos problemas se evitan por completo, ya que Java provee la recolección automática de basura.

El sistema de archivos

La persona que invoca a un applet de Java dentro de un navegador Web puede controlar el acceso de ese applet a los archivos, para lo cual especifica una opción al navegador. El usuario puede evaluar la confiabilidad del applet y decidir (por ejemplo) que puede leer un archivo local, pero no escribir en un archivo local. Las opciones predeterminadas no proveen acceso a los archivos.

Resumen de seguridad

Java provee varios mecanismos de protección:

- El lenguaje y el compilador, por ejemplo, ausencia de apuntadores.
- Verificación del código de bytes.
- El cargador de clases.
- El sistema de archivos y el acceso a redes.

Las medidas de seguridad de Java son extensas y de amplio rango. Sin embargo, no existe una garantía absoluta de que un programador malicioso no pueda penetrar las defensas.

Código fuente abierto

El movimiento de código fuente abierto se basa en el concepto de que el código fuente del software que utilizamos debe estar disponible. Entre otras cosas, esto evita los estándares de propiedad (pertener a una sola empresa). Cabe mencionar que el software de código fuente abierto no necesita ser gratis (sin costo), sino que debe ser libre en el sentido de ser abierto (que se pueda ver).

El movimiento de código fuente abierto tuvo problemas con Java hasta el año 2006, porque en teoría todas las herramientas (como los compiladores) utilizadas para crear software de código fuente abierto deberían ser también de código fuente abierto. Java no lo era, aun cuando se ofrecía sin costo. El resultado fue que muchos programadores de código fuente abierto no usaban Java.

Sin embargo, en el año 2006 Sun anunció que iba a convertir a Java en un producto de código fuente abierto. Esto no fue tan simple, ya que el proyecto de Java en sí utilizaba cierto código propietario. En 2008, el 99% de las herramientas y bibliotecas de clases de Java se podían clasificar como código fuente abierto en el proyecto OpenJDK de Sun.

Las versiones de Java

La versión actual que se utilizó en este libro es Java SE 6, conocida simplemente como la versión 6. Sun provee distintas categorías del sistema Java:

1. Standard Edition (SE). Probablemente la versión que usted utilizará para este libro.
2. Enterprise Edition (EE), para las organizaciones (potencialmente grandes) que requieren de los principales tipos de software de servidor, bases de datos distribuidas, ventas por Internet, etc.
3. Mobile Edition (ME). Esta edición es capaz de ejecutarse en una pequeña cantidad de memoria; está orientada a la incrustación de código en dispositivos tales como teléfonos móviles, bípers y decodificadores.
4. Tarjeta de Java. Se ejecuta en muy poca memoria, está orientada a las tarjetas inteligentes.

Herramientas de Java

Por lo general, las mejoras a Java surgen mediante la adición de nuevas bibliotecas (gracias a la orientación a objetos de Java) y también por medio de software que permite usar a Java en nuevas formas (por ejemplo, RMI). Aquí vamos a inspeccionar las herramientas más importantes que proporciona Sun en forma de bibliotecas y tecnologías. Sin embargo, tenga en cuenta que hay también muchas otras empresas que crean software de Java.

Bibliotecas de Java

Java es un lenguaje pequeño, por lo que la mayor parte del poder de los programas de Java surge de las herramientas que proporcionan las grandes (y algunas veces complejas) bibliotecas. Se ha realizado un esfuerzo muy grande por agregar estas bibliotecas para promover el uso de Java. A continuación le presentamos una selección de la lista completa:

- Java Language and Utilities. Proveen las herramientas (descritas en este libro) para uso de ventanas y E/S de archivos. También cuentan con soporte para conectividad de redes: escribir programas que se comuniquen con los programas en otra computadora.
- Java Electronic Commerce Framework. La carrera por comercializar Internet ha comenzado y esta biblioteca puede ser útil, ya que consta de herramientas para enviar información de tarjetas de crédito o dinero electrónico en forma segura por Internet.
- Java Security. Provee herramientas para las firmas digitales y la autenticación.
- Java Media Framework. Herramientas completas para reproducción de medios.
- Java Collections Framework. Estructuras de datos como conjuntos, listas enlazadas y mapas.
- Java 3D. Clases para crear y mostrar mundos tridimensionales.

Java beans

Sun afirma que un bean es “un componente de software reutilizable que se puede manipular de manera visual en una herramienta de desarrollo”. Antes de explicar el significado de esto, vamos a examinar la industria de los electrodomésticos. Existen estándares establecidos que aseguran que, por ejemplo, su nuevo reproductor de CD trabajará de manera apropiada con cualquier amplificador que cuente con entradas para CD. El estándar está al nivel de los tamaños de los conectores y el

nivel del voltaje. El resultado de esto es que es muy simple crear un nuevo sistema de alta fidelidad, ya que sólo hay que conectar un conjunto de módulos.

Regresemos al software. Imagine que está escribiendo un editor de texto en Java y que desea integrar un corrector ortográfico. ¿Qué pasaría si pudiera abrir su procesador de palabras favorito y arrastrara el botón de su corrector ortográfico hacia su editor de texto? (En realidad no lo puede hacer en estos momentos, ¡pero es una excelente idea!)

El concepto de los beans es bastante similar. Un programador puede escribir beans, que son clases de Java escritas en un estilo específico. Un bean podría proporcionar una GUI sofisticada para el programador (por ejemplo, un bean de hoja electrónica de cálculo) o su importante característica podría estar en el código oculto y los datos (por ejemplo, un bean para la corrección ortográfica). En cualquier caso, el bean proporciona un ícono en tiempo de diseño, para que el programador pueda incorporarlo y manipularlo en forma visual.

He aquí el proceso del bean con más detalle:

- Un bean se diseña y codifica o se compra en algún otro lado. El proceso de codificar un bean es similar a la codificación de cualquier clase de Java, pero el programador tiene que seguir ciertas convenciones de nomenclatura y estilo.
- El software en el que se va a colocar el bean está escrito de tal forma que permita la incorporación del bean. En algunos casos simples no se requiere codificación.
- Para incorporar el bean a la aplicación utilizamos un entorno de desarrollo de beans especial. Esto permite seleccionar beans en forma gráfica para colocarlos en una aplicación o conectarlos a otros beans.

Por lo tanto, los beans son clases de Java escritas de cierta forma específica. Se pueden incorporar con facilidad en el software listo para beans. El sistema .NET tiene una herramienta similar llamada “Controles de usuario”, en donde se puede incorporar código empaquetado con facilidad en un programa.

Bases de datos – JDBC

Las bases de datos son un gran negocio. Sin una base de datos, los bancos, las empresas de fabricación, las aerolíneas, los supermercados, etcétera, no podrían funcionar. Aparte del uso de los procesadores de palabras, las hojas de cálculo y el correo electrónico, la base de datos se puede considerar como el componente de software más vital en los negocios.

Como ejemplo, una tienda que vende computadoras podría utilizar una base de datos para mantener los niveles de existencias, las direcciones de clientes y proveedores, el envío masivo de correo y la elaboración de facturas, además de los salarios del personal.

La mayoría de las bases de datos se denominan “relacionales”. Los datos se mantienen en tablas que se pueden relacionar entre sí. Se puede acceder en forma local a la mayoría de las bases de datos, ya sea mediante comandos de menú o consultas desde el teclado, en un lenguaje estándar conocido como SQL (Lenguaje de Consulta Estructurado).

Con frecuencia, las organizaciones grandes cuentan con una base de datos central, a la que tienen acceso un gran número de personas, a menudo desde distintos países y a través de distintos tipos de computadoras. Aquí es donde reditúa la portabilidad de Java. Las clases JDBC (Conectividad de Bases de datos de Java) permiten a un programa de Java que se ejecuta en (por ejemplo) un equipo de escritorio pasar consultas SQL a una base de datos que se encuentra en otra computadora. Después se devuelven los resultados.

Desde luego que existe otro software que puede hacer esto, pero los defensores de Java recalcan su portabilidad en este escenario.

Para resumir, JDBC permite el acceso a las bases de datos compatibles con SQL.

Java e Internet

Internet es una colección de redes de computadoras a nivel mundial, junto con el software y los protocolos que les permiten interactuar. Uno de estos protocolos permite mostrar páginas Web e hipervínculos en un navegador Web. A esto se le conoce como World Wide Web, pero también hay otros protocolos como los que se utilizan para el correo electrónico y la transferencia de archivos (FTP).

En esta sección analizaremos el soporte de Java para la programación en Internet (sin un navegador Web) y hablaremos sobre los sitios Web por separado.

Java es un lenguaje muy relacionado con Internet y las intranets (una intranet es una red sostenida dentro de una organización, que utiliza tecnología de Internet). Es un lenguaje que permite desempeñar con facilidad muchas actividades relacionadas con Internet. En el capítulo 17 sobre archivos vimos que un programa puede leer datos de cualquier archivo en Internet (siempre y cuando tenga los privilegios de seguridad apropiados). Esto significa que Internet se ve como un enorme sistema de archivos global. Un programa de Java puede acceder a la información en cualquier parte, con la misma facilidad que si estuviera en un disco local. Puede recuperar y mostrar información textual, gráfica, de animaciones y audio. Se pueden colocar pedidos de productos, para lo cual se agrega información a un archivo remoto (siempre y cuando las medidas de seguridad lo permitan).

Java es un lenguaje mundial en un sentido distinto. Utiliza el estándar Unicode para representar y manipular caracteres. Éste es un estándar internacional que provee 16 bits para la representación de un carácter, en contraste a los estándares anteriores como ASCII que sólo proveen 8 bits. Esto significa que los programas de Java pueden manipular caracteres de todos los lenguajes del mundo, incluyendo (por ejemplo) el mandarín, urdú y japonés. Esto también significa que los programadores pueden usar su propio lenguaje para los nombres de las variables, los métodos y las clases.

Por último, pero no por ello menos importante, es posible que partes de un programa de Java residan en distintos lugares en Internet, para recuperarlas y ejecutarlas según sea necesario. Como hemos visto, los programas de Java se construyen a partir de clases. El sistema de Java utiliza la vinculación dinámica. Esto significa que una clase sólo se carga y vincula con el objeto que la utiliza cuando se hace la primera referencia a ella. Por lo general, esta primera referencia es la creación de una instancia de la clase utilizando el operador `new` y el método constructor de la clase. Las distintas clases que componen a un programa se pueden almacenar en distintas computadoras a través de Internet. Al requerirse una clase, se puede recuperar, cargar, vincular y finalmente ejecutar. No siempre se da el caso de que un usuario llegue a necesitar esta capacidad, pero puede ser muy útil.

RMI – Invocación de Métodos Remotos

Hace unos cuantos años, el propietario de una PC doméstica no estaba expuesto a la posibilidad de interactuar entre varias computadoras. Ahora, con el increíble aumento en el uso de Internet, la idea de interactuar con programas en otras máquinas es muy popular. De igual forma, en el área de cómputo de las empresas, las redes son la norma.

En ciertas áreas de aplicación puede ser conveniente distribuir el poder de procesamiento entre varias computadoras, para que después interactúen entre sí cuando se requiera. Imagine una empresa multinacional. En definitiva, debe contar con una extensa base de datos y podría usar la biblioteca de JDBC para acceder a ella. Pero si el área de aplicación no es estándar, entonces tal vez una base

de datos no sea apropiada. ¿Qué pasaría si la empresa quisiera proveer software a un grupo de participantes en distintos países para cooperar en la creación de un diseño? Para ello se podría requerir una ventana en la que cualquiera pudiera escribir o dibujar. Cada participante podría ver el trabajo de los demás. Un posible arreglo sería que cada participante tenga un programa en su computadora para manejar el dibujo local y que tenga un programa en un servidor central para transmitir cualquier entrada nueva. El software de este tipo se puede desarrollar con más facilidad si la computadora de un individuo puede invocar los métodos en un servidor. En Java, a esto se le conoce como RMI (Invocación de Métodos Remotos).

Ya hay una metodología estándar para este tipo de software distribuido, conocida como CORBA (Arquitectura Común de Intermediarios en Peticiones a Objetos), la cual no está enlazada con ninguna empresa o lenguaje de programación en específico. Microsoft cuenta con un sistema similar, conocido como DCOM (Modelo de Objetos de Componentes Distribuidos). Al momento de escribir este libro no estaba claro cuál de las dos metodologías sería la dominante.

Entonces, ¿qué es lo que RMI nos deja hacer? En esencia, usted (o alguien más) puede configurar software de Java en un servidor. Después puede escribir software de Java en otro equipo, el cual puede llamar a los métodos en el servidor. Además, se pueden pasar parámetros como en la invocación a métodos normales y parte del poder de la RMI proviene del hecho de que se pueden pasar parámetros de cualquier tipo, no sólo simples números o cadenas de texto.

RMI tiene mucho potencial, ya que es más simple que sus competidores, pero son las fuerzas de mercado y no las cuestiones técnicas lo que determinará quién será el ganador en el mundo de la computación distribuida.

Java y World Wide Web

WWW se refiere a la colección de páginas Web que están disponibles por medio de un navegador Web. También soporta hipervínculos entre las páginas. En WWW, Java se utiliza para escribir applets (programas que se ejecutan en una página Web) y también para proveer tecnologías de software para crear sitios Web interactivos que implican una cantidad considerable de codificación en el servidor.

Applets

Al igual que para escribir aplicaciones (lo que vimos en todo el libro), podemos usar Java para escribir *applets*, los cuales se compilan y almacenan en un servidor Web junto con las páginas Web convencionales. Dentro de una página Web convencional es posible emitir instrucciones para descargar y ejecutar un applet. Cabe mencionar que el applet se ejecuta en la computadora del usuario, dentro del navegador Web, por lo que las cuestiones de seguridad (que vimos en un capítulo anterior) son cruciales. Para que los applets se utilizaran ampliamente, Sun tuvo que hacer que las organizaciones que escribían navegadores incorporaran una JVM, lo cual hicieron. Aunque los applets han estado ofreciendo una experiencia Web dinámica por muchos años, ahora tienen que competir con DHTML y Adobe Flash.

Servlets de Java y JSP

Para habilitar la interacción Web, los programas en una computadora servidor interactúan con el software en una computadora cliente (por lo general, un navegador Web). Por ejemplo, un usuario

podría introducir un nombre en un campo de una página Web y estos datos se pasarían a un programa en el servidor, que podría usar este nombre para acceder a una base de datos. Hay varias formas de crear software del lado servidor, siendo la más común el uso de la herramienta de Interfaz de Puerta de Enlace Común (CGI: Common Gateway Interface) de HTTP/HTML, que por lo general se maneja mediante un programa de Perl en el servidor, aunque los programas CGI se pueden escribir en casi cualquier lenguaje, incluyendo C, C++ y Java.

Los servlets pueden simplificar la tarea de programación. Un servlet es un programa de Java codificado de acuerdo con ciertas convenciones, con clases utilitarias que son de mucha utilidad (por ejemplo, para acceder a los datos del cliente). Muchos servidores (como Apache y Microsoft IIS) permiten ejecutar servlets.

Los beneficios de los servlets son:

- Cada vez que se utiliza la página Web y se envían datos al servidor, el servlet individual crea un nuevo hilo. En la metodología CGI es común el caso en que se crea y ejecuta una copia fresca del programa de CGI. Esto se considera ineficiente.
- Un servlet se puede ejecutar en una caja de arena (sandbox) para proveer una mejor seguridad.
- Un servlet es independiente de la plataforma (suponiendo que el software de servidor existente permita incorporar servlets).

Además de los servlets se pueden usar Páginas de Servidor de Java (JSP). Una JSP permite separar el diseño de páginas Web de la codificación de Java, para que los especialistas se puedan hacer cargo de estas tareas.

Las diferencias clave entre servlets y JSP son:

- Un servlet es un programa de Java que (al ejecutarse) crea el texto de una página Web. Si analiza el código de un servlet verá que contiene muchas instrucciones “print” para generar el código de HTML.
- Una JSP es una página Web con código de Java incrustado en su interior. Las etiquetas especiales en la página permiten ejecutar objetos de Java. La intención es ayudar al experto en HTML a incorporar clases previamente escritas sin necesidad de entrar en los detalles de la programación.

● La oposición: plataforma .NET de Microsoft

En la actualidad, los sistemas operativos Windows de Microsoft representan cerca del 90% del mercado mundial de los sistemas operativos para PC. Éste ha sido considerado casi un monopolio masivo. Los sistemas operativos Microsoft se ejecutan en procesadores Intel; a esta poderosa combinación se le conoce algunas veces como Wintel. Las demás empresas en la industria de las computadoras están celosas de este poder y les gustaría quebrantar el supuesto monopolio. Ven a Microsoft como la empresa que domina el mercado, lo cual debilita su participación en el mismo y en un momento dado puede dejarlas fuera del negocio. Java se ha convertido en un arma para esta batalla. Encabezados por Sun Microsystems, los oponentes de Microsoft han promovido a Java como el medio para crear software portable que se ejecute en cualquier máquina y bajo cualquier sistema operativo.

Microsoft tiene un producto similar a Java, conocido como plataforma .NET, el cual se liberó siete años después de Java, en el año 2002. Éstas son las características principales:

- Tiene una enorme biblioteca de clases.
- Está muy orientado a Internet.

- Cuenta con varios lenguajes totalmente OO, incluyendo una nueva versión de Visual Basic y un nuevo lenguaje conocido como C# (C Sharp). C# es muy similar a Java.
- Tiene un IDE estándar para todos sus lenguajes.
- Los programas escritos para .NET se compilan en código de bytes de Microsoft y son interpretados por el CLR (componente de Lenguaje común en Tiempo de Ejecución), el cual corresponde a la JVM de Java. Aunque en la actualidad .NET sólo se ejecuta en sistemas operativos Microsoft, es posible portar todo el sistema a otros sistemas operativos.
- Hay una versión reducida para los sistemas de equipos de bolsillo.

Es difícil comparar las plataformas Java y .NET, pero al programador principiante le pueden interesar los siguientes puntos:

- En .NET las aplicaciones de escritorio (como las que vimos en este libro) se desarrollan en el IDE de Microsoft. Creemos que es más simple para los principiantes que el IDE Eclipse.
- En cuanto a la creación de sitios Web en los que se involucra mucho código de servidor, las herramientas de Microsoft proveen un enfoque más visual, con lo cual se reduce la codificación de HTML. Sin embargo, hay un debate para determinar si esto aplica a la hora de crear sistemas extensos. En general, las dos plataformas pueden realizar tareas similares; además una organización de gran tamaño considerará otros factores tales como el mantenimiento, la cantidad de personal requerida y el hardware/software de servidor.

● JavaScript

JavaScript es un lenguaje de secuencias de comandos que se utiliza en las páginas Web. La razón de su nombre es que Netscape (quien produjo uno de los primeros navegadores Web) escuchó sobre la creación de Java y le puso ese nombre a su lenguaje de secuencia de comandos. Pero en realidad no hay mucha similitud entre los lenguajes y su área de aplicación.

He aquí las principales diferencias:

- El código de JavaScript está incrustado en las páginas Web de HTML; si descarga una página Web con JavaScript habilitado, podrá ver el código de JavaScript.
- Aparentemente se ve igual a Java en términos de las estructuras de control (palabras clave similares, signos de llaves) pero no permite crear clases ni herencia. Se considera “basado en objetos” y no totalmente orientado a objetos.
- Es un lenguaje débilmente tipificado. Los nombres de las variables no se necesitan declarar antes de usarlas, e incluso si las declaramos no podemos especificar su tipo.
- Sólo se puede ejecutar dentro de un navegador Web y sólo puede acceder a los elementos del navegador, como los controles de los formularios de HTML y los movimientos del ratón. Por lo tanto, se puede utilizar para crear páginas con una apariencia visual interesante y para validar las entradas en los campos de texto de HTML. No puede acceder a los archivos y no se puede utilizar para escribir aplicaciones o applets.

Tal vez sería conveniente que aprendiera sobre JavaScript después de Java. Esto le será relativamente simple, ya que ahora comprende los conceptos de variables, estructuras de control, métodos y la notación “punto” de un objeto/método. JavaScript tiene todos estos elementos, pero los usa de una manera ligeramente distinta.

● Conclusión

Hay varias tendencias claras en la industria de las computadoras:

- Un tremendo aumento en el uso de Internet.
- La importancia de la interacción y los servicios Web.
- El uso de intranets (una intranet es una red local, por lo general dentro de una empresa, que utiliza tecnología de Internet).
- Una reducción en el costo de las redes.
- La fusión de las industrias de las computadoras y las telecomunicaciones.
- La incorporación (incrustación) de computadoras dentro de productos electrodomésticos.
- El interés sobre multimedia.
- La aceptación de la necesidad vital de trabajar con las computadoras en forma segura entre redes.
- La lucha contra el dominio de Microsoft (sistema operativo Windows) e Intel (PC).
- La introducción del sistema .NET de Microsoft.

Java surgió de la necesidad de crear nuevas tecnologías para hacer frente a estas tendencias. Java no es una solución milagrosa que resuelve los problemas como por arte de magia. No hay una “aplicación definitiva” que haga de Java algo indispensable, así como la hoja de cálculo VisiCalc hizo de la computadora Apple II un éxito. En vez de ello, Java combina lo mejor de las tecnologías actuales en una síntesis muy elegante para proveer una “navaja suiza” de herramientas para trabajar con las computadoras en el nuevo milenio.

Resumen

Muchos de los temas anteriores podrían llenar todo un libro de manera individual; también están las áreas de seguridad y redes que no tratamos en este libro. Hay nuevos productos de Java que salen rápidamente al mercado y a menudo son de consideración. Le recomendamos visitar con frecuencia el sitio Web de Sun para novedades de productos y descargas. El URL es: <http://java.sun.com/>.

● Ejercicios

- 26.1** “Java es sólo otro lenguaje más de programación” (Bill Gates). Exprese su opinión al respecto.
- 26.2** La sobrecarga en rendimiento de los programas de Java hace que no sean prácticos para muchas aplicaciones. Exprese su opinión al respecto.
- 26.3** Examine la diferencia entre C# y Java, en términos de las herramientas que ofrece cada lenguaje.
- 26.4** El lenguaje C# de Microsoft acabará con Java. Exprese su opinión al respecto.
- 26.5** Sugiera el futuro para Java.

APÉNDICE A



Bibliotecas de Java

En este apéndice se describen clases y métodos selectos de las bibliotecas de Java para fines de referencia e investigación. Todos los programas de Java utilizan algunas clases de las bibliotecas para lograr su tarea. Java en sí es un lenguaje pequeño, por lo que las bibliotecas proveen la mayor parte de la funcionalidad necesaria. Java cuenta con una variedad de métodos agrupados en clases, que a su vez están organizadas en varias bibliotecas (paquetes). Las clases que se listan aquí son todas las que utilizamos en el libro. Son más que suficientes para escribir una amplia variedad de programas. Las listamos en orden alfabético; las que empiezan con la letra **J** están en el orden que les correspondería sin la **J**.

Proporcionamos el encabezado completo de cada uno de los métodos que se encuentran en las bibliotecas. Por ejemplo:

```
public void drawLine(int x1, int y1, int x2, int y2)
```

de tal forma que se tenga a la mano la información sobre cómo utilizarlos. Al inspeccionar todo el encabezado podemos ver qué parámetros se requieren (o los paréntesis vacíos, en caso de que no se requieran parámetros) y el valor de retorno a devolver (o **void** si no se devuelve ningún valor). El siguiente es un comentario sobre las bibliotecas (recuerde que un campo puede ser una variable o un método):

- Casi todos los campos disponibles de las bibliotecas son métodos; hay muy pocas variables. Esto refleja el principio de diseño del ocultamiento de información, en donde el acceso a las variables por lo general se realiza mediante la llamada a un método.
- Todos los campos se identifican como **public**, ya que sólo se pueden utilizar los elementos públicos.
- Los campos identificados como **static** pertenecen a una clase como un todo y no a los objetos individuales que se instancian a partir de la clase. La notación para utilizar estos campos es:

```
NombreClase.nombreMétodo();
```

Por ejemplo:

```
Integer.toString(entero);
```

- Los métodos constructores, como `JTextField` y `JLabel`, no tienen tipo de valor de retorno.

La instrucción `import` es una forma de permitir al programador abreviar el nombre de una clase o un método, en vez de tener que proporcionar su nombre completo. Un nombre completo es largo y difícil de manejar, ya que incluye el nombre del paquete. Por ejemplo, el nombre completo de la clase `JButton` es `javax.swing.JButton`. Podemos usar sólo el nombre `JButton` abreviado, siempre y cuando aparezca la siguiente declaración:

```
import javax.swing.JButton;
```

en el encabezado del programa. En todos los casos explicamos qué instrucción `import` se requiere (en caso de ser necesario) para usar la clase de una manera conveniente.

El paquete `java.lang` se importa de manera automática en todos los programas de Java, ya que provee métodos vitales que se utilizan en casi todos los programas de Java. Por ende, podemos usar las clases de este paquete sin necesidad de la instrucción `import`.

● ArrayList

Estructura de datos que permite agregar y eliminar elementos. La estructura se expande y contrae según se requiera.

<code>import java.util.ArrayList;</code>	
<code>public ArrayList <NombreClase>()</code>	Crea un nuevo objeto <code>ArrayList</code> para que contenga objetos de la clase <code>NombreClase</code> .
<code>public void add(int índice, NombreClase elemento)</code>	Inserta el elemento especificado en la posición especificada. Desplaza el elemento que se encuentra en esa posición y cualquier elemento subsiguiente (suma 1 a sus valores de índice).
<code>public boolean add(NombreClase elemento)</code>	Adjunta el elemento especificado al final del objeto <code>ArrayList</code> . Devuelve <code>true</code> .
<code>public void clear()</code>	Elimina todos los elementos del objeto <code>ArrayList</code> .
<code>public boolean contains (NombreClase elemento)</code>	Devuelve <code>true</code> si el objeto se encuentra en el objeto <code>ArrayList</code> .
<code>public NombreClase get(int índice)</code>	Devuelve el elemento en la posición especificada.
<code>public void remove(int índice)</code>	Elimina el elemento en la posición especificada.
<code>public NombreClase set(int índice, NombreClase elemento)</code>	Sustituye el elemento en la posición especificada con el elemento especificado. Devuelve el objeto que se encontraba en esta posición.
<code>public int size()</code>	Devuelve el número de elementos en el objeto <code>ArrayList</code> .

● BorderLayout

La mayor parte de los programas en este libro utilizan el esquema o diseño de flujo para crear una interfaz de usuario. En esta metodología, los componentes se agregan en la pantalla en orden de izquierda a derecha. Si no se llena por completo una fila de componentes, los componentes restantes se centran.

Sin embargo, existen otras metodologías. Considere un procesador de palabras o un paquete de dibujo ordinarios; hay una gran área de trabajo central, con componentes (botones, etc.) distribuidos a través de la parte superior y/o lateral. Cuando el usuario amplía la ventana, el espacio central se expande pero el espacio asignado a los botones permanece fijo. Para crear esto en Java utilizamos el esquema de bordes. He aquí un fragmento de código que crea un esquema de bordes, con una gran área de dibujo y dos botones. Observe que establecimos el esquema de la ventana a `BorderLayout`.

```
private void crearGUI() {
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    Container ventana = getContentPane();
    ventana.setLayout(new BorderLayout());
    panelBotones = new JPanel();

    panelDibujo = new JPanel();
    panelDibujo.setBackground(Color.white);

    botón1 = new JButton("botón1");
    panelBotones.add(botón1);
    botón1.addActionListener(this);

    botón2 = new JButton("botón2");
    panelBotones.add(botón2);
    botón2.addActionListener(this);

    ventana.add("South", panelBotones);
    ventana.add("Center", panelDibujo);
}
```

El esquema de bordes opera de la siguiente manera:

La ventana se considera compuesta de cinco áreas: un área central y franjas en cada borde. Al llamar al método `add` para un esquema de bordes, especificamos el área con una cadena: `"North"`, `"South"`, `"East"`, `"West"`, `"Center"`.

Sólo se puede agregar un elemento a cada área. Para poder agregar los dos botones, creamos un panel (llamado `panelBotones` en este ejemplo). Agregamos cada botón al panel y después agregamos el panel a un borde de la ventana (`"South"` en este ejemplo). Después agregamos el panel de dibujo al centro. Si se requiere, podemos agregar elementos en todos los bordes de la ventana.

Si analiza con cuidado el código podrá ver que no hemos especificado un tamaño para el panel de dibujo; entonces, ¿qué tan grande es? La respuesta es que se agregan botones de tamaño estándar al borde y después se expande el área central para ocupar el resto del espacio.

Además, este esquema se recalcula si el usuario cambia el tamaño de la ventana. La figura A.1 muestra dos pantallas del esquema anterior, con un cambio de tamaño.

El esquema de bordes es un poco más complicado, pero es bueno en algunas situaciones.



Figura A.1 Esquema de bordes (se muestran dos tamaños de ventana).

● JButton

Un botón de GUI. Al hacer clic en el botón se crea un evento:

```
import javax.swing.JButton; O import javax.swing.*;
```

```
public JButton(String etiqueta)
```

Crea un botón con la etiqueta.

```
public void addActionListener  
    (Object objeto)
```

Registra el objeto para manejar el evento
(por lo general **this**).

```
public void setFont(Font f)
```

Establece el tipo y tamaño de fuente. Ejemplo:
**button.setFont(new Font(null,
Font.BOLD, 60));**
Para el constructor de **Font**, el segundo parámetro
es el estilo. Las opciones son **Font.BOLD**,
Font.ITALIC, **Font.PLAIN**. El tercer parámetro
es el tamaño de la fuente.



*JButton continúa*

Programa de ejemplo (vea la figura A.2):

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class DemoBotón extends JFrame implements ActionListener {

    private JButton botón;
    private JTextField campoTexto;

    public static void main(String[] args) {
        DemoBotón marco = new DemoBotón();
        marco.setSize(400,300);
        marco.crearGUI();
        marco.setVisible(true);
    }

    private void crearGUI() {
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        Container ventana = getContentPane();
        ventana.setLayout(new FlowLayout());

        botón = new JButton("Haga clic");
        ventana.add(botón);
        botón.addActionListener(this);

        campoTexto = new JTextField(10);
        ventana.add(campoTexto);
    }

    public void actionPerformed(ActionEvent event) {
        campoTexto.setText("hizo clic en el botón");
    }
}
```

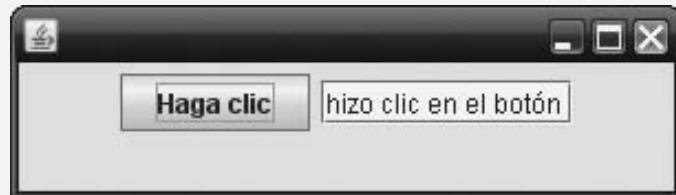


Figura A.2 Un botón y un campo de texto.

● Calendar

<code>import java.util.Calendar;</code>	
<code>public static Calendar getInstance()</code>	Devuelve una instancia de la clase <code>Calendar</code> .
<code>public int get(Calendar.DAY_OF_MONTH)</code>	Devuelve el día del mes.
<code>public int get(Calendar.DAY_OF_WEEK)</code>	Devuelve el día de la semana. 0 es domingo.
<code>public int get(Calendar.DAY_OF_YEAR)</code>	Devuelve el día del año.
<code>public int get(Calendar.HOUR_OF_DAY)</code>	Devuelve la hora del día, entre 0 y 23, en el sistema de reloj de 24 horas. 0 es medianoche.
<code>public int get(Calendar.MINUTE)</code>	Devuelve el número de minutos después de la hora.
<code>public int get(Calendar.MONTH)</code>	Devuelve el mes. 0 es enero.
<code>public int get(Calendar.SECOND)</code>	Devuelve el número de segundos después del minuto.
<code>public int get(Calendar.MILLISECOND)</code>	Devuelve el número de milisegundos después del segundo.
<code>public int get(Calendar.YEAR)</code>	Devuelve el año.

Código de ejemplo:

```
Calendar calendario = Calendar.getInstance();
int hora = calendario.get(Calendar.HOUR_OF_DAY);
campoTexto.setText("la hora es " + hora);
```

Vea también `currentTimeMillis` en la clase `java.lang.System`.

● JCheckBox

Una casilla de verificación es un componente de GUI que se agrega a un marco, de la misma forma que un botón. Cuando el usuario hace clic en una casilla, se crea un evento y se maneja a través del método `itemStateChanged`.

```
import javax.swing.JCheckBox; o import javax.swing.*;
```

<code>public JCheckBox(String s)</code>	Crea una casilla de verificación con la etiqueta <code>s</code> .
<code>public void addItemClickListener (Object objeto)</code>	Registra el objeto para manejar eventos.
<code>public boolean isSelected()</code>	Devuelve <code>true</code> si se seleccionó la casilla.



*JCheckBox continúa*

Programa de ejemplo (vea la figura A.3):

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class DemoCheckBox extends JFrame implements ItemListener {

    private JCheckBox refresco, hamburguesa, papasfritas;
    private JTextArea áreaTexto;

    public static void main(String[] args) {
        DemoCheckBox marco = new DemoCheckBox();
        marco.setSize(400,300);
        marco.clearGUI();
        marco.setVisible(true);
    }

    private void clearGUI() {
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        Container ventana = getContentPane();
        ventana.setLayout(new FlowLayout());
        refresco = new JCheckBox("refresco");
        ventana.add(refresco);
        refresco.addItemListener(this);

        hamburguesa = new JCheckBox("hamburguesa");
        ventana.add(hamburguesa);
        hamburguesa.addItemListener(this);

        papasfritas = new JCheckBox("papasfritas");
        ventana.add(papasfritas);
        papasfritas.addItemListener(this);

        áreaTexto = new JTextArea(5,3);
        ventana.add(áreaTexto);
    }

    public void itemStateChanged(ItemEvent event) {
        String nuevaLínea = "\r\n";
        áreaTexto.setText("");
        if (refresco.isSelected()) {
            áreaTexto.append("se seleccionó refresco" + nuevaLínea);
        }
        if (hamburguesa.isSelected()) {
            áreaTexto.append("se seleccionó hamburguesa" + nuevaLínea);
        }
        if (papasfritas.isSelected()) {
            áreaTexto.append("se seleccionó papasfritas" + nuevaLínea);
        }
    }
}
```

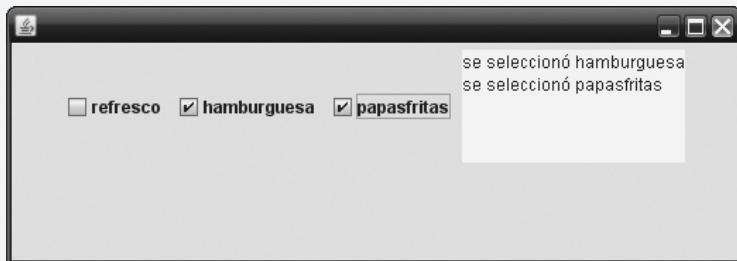


Figura A.3 Casilla de verificación.

● JComboBox

Un cuadro combinado es una lista desplegable de cadenas de GUI (figura A.4). El usuario puede seleccionar un elemento al hacer clic en él. Cuando el usuario hace clic en un elemento, se crea un evento. El manejador de eventos puede determinar el número de índice o la cadena en la que se hizo clic. Los valores de los índices empiezan en 0.

<code>public JComboBox()</code>	Crea un nuevo cuadro combinado.
<code>public void addItem(Object objeto)</code>	Agrega al cuadro combinado un nuevo elemento después de los valores actuales.
<code>public void insertItemAt(Object objeto, int índice)</code>	Inserta un nuevo elemento en la posición del índice especificado. Los elementos anteriores se mueven hacia abajo.
<code>public void removeItemAt(int índice)</code>	Elimina el elemento en la posición del índice especificado. Se cierra el hueco.
<code>public void removeAllItems()</code>	Elimina todos los elementos.
<code>public int returnItemCount()</code>	Devuelve un conteo del número de elementos.
<code>public void addActionListener(Object objeto)</code>	Registra el objeto para manejar eventos.
<code>public Object getSelectedItem()</code>	Devuelve el objeto seleccionado.
<code>public int getSelectedIndex()</code>	Devuelve el índice del objeto seleccionado. Devuelve -1 si no hay un elemento seleccionado.

Programa de ejemplo:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
```



*JComboBox continúa*

```
public class DemoCombo extends JFrame implements ActionListener {  
    private JComboBox combo;  
    private JTextField campoTexto;  
    private JButton botón;  
  
    public static void main(String[] args) {  
        DemoCombo marco = new DemoCombo();  
        marco.setSize(400,300);  
        marco.crearGUI();  
        marco.setVisible(true);  
    }  
  
    private void crearGUI() {  
        setDefaultCloseOperation(EXIT_ON_CLOSE);  
        Container ventana = getContentPane();  
        ventana.setLayout(new FlowLayout());  
  
        combo = new JComboBox();  
        combo.addItem("rojo");  
        combo.addItem("azul");  
        combo.addItem("amarillo");  
  
        combo.addActionListener(this);  
        ventana.add(combo);  
  
        campoTexto = new JTextField(16);  
        ventana.add(campoTexto);  
        botón = new JButton("insertar");  
        botón.addActionListener(this);  
        ventana.add(botón);  
    }  
  
    public void actionPerformed(ActionEvent event) {  
        if (event.getSource() == botón) {  
            insertar();  
        }  
        else {  
            int indice = combo.getSelectedIndex();  
            String elemento = (combo.getSelectedItem()).toString();  
            campoTexto.setText("elemento " + Integer.toString(indice)  
            + ", " + elemento + " seleccionado");  
        }  
    }  
    public void insertar(){  
        combo.insertItemAt("NEGRO", 2);  
    }  
}
```

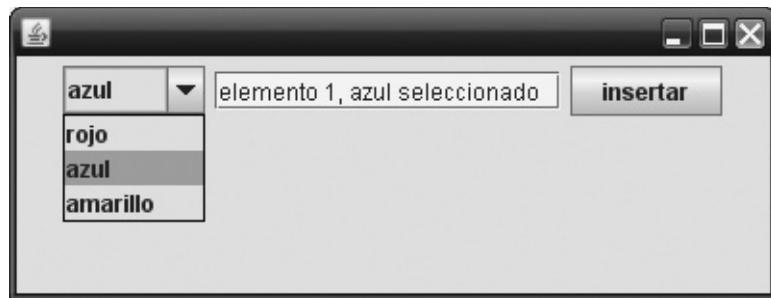


Figura A.4 Cuadro combinado.

● Container

Un contenedor es un componente que puede contener otros componentes de GUI.

<code>import java.awt.*;</code> o <code>import java.awt.Container;</code>	
<code>public void add(Component c)</code>	Agrega un componente a este contenedor.
<code>public void setLayout(LayoutManager m)</code>	Establece el administrador de esquemas para este componente. Vea por ejemplo <code>FlowLayout</code> .

En la mayoría de los programas de este libro, un contenedor se crea a partir de un objeto marco y se utiliza para contener componentes de GUI tales como los botones. La mayoría de los programas de este libro hacen esto:

```
Container ventana = getContentPane();
ventana.setLayout(new FlowLayout());
ventana.add(botón);
```

● DataInputStream

```
import java.io.DataInputStream; O import java.io.*;
```

<code>public DataInputStream(String nombre)</code>	Crea un nuevo flujo de datos de entrada.
<code>public final String readline()</code>	Lee la siguiente línea de texto del flujo de datos.

● DecimalFormat

Provee el formato de los números a mostrar en pantalla.

<code>import java.text.DecimalFormat;</code>	
<code>public DecimalFormat(String patrón)</code>	Constructor. Crea un objeto <code>DecimalFormat</code> que utilizará <code>patrón</code> .
<code>public String format(double d)</code>	Convierte el valor <code>double</code> en la cadena equivalente, de acuerdo con el patrón.
<code>public String format(int i)</code>	Convierte el valor <code>int</code> en la cadena equivalente, de acuerdo con el patrón.

La siguiente tabla sintetiza los caracteres de formato que se pueden utilizar para crear patrones:

Carácter	Significado
#	inserta un dígito si hay uno
0	siempre inserta un dígito
,	inserta una coma
.	inserta un punto decimal
E	inserta E seguida de la potencia de 10
\$	inserta un signo de dólares

Ejemplo:

```
double d = 12.34;
DecimalFormat formato = new DecimalFormat("###.##");
campoTexto.setText(formato.format(d));
```



● Double

Está en `java.lang`. No se requiere instrucción `import`.

<code>public static String parseDouble(String s)</code>	Convierte una cadena en un valor <code>double</code> . Se crea una excepción <code>NumberFormatException</code> si la cadena no contiene caracteres válidos.
<code>public static String toString(double d)</code>	Convierte el valor <code>double</code> en la cadena equivalente.

Código de ejemplo:

```
double d = Double.parseDouble("0.123");
String s = Double.toString(d);
```

● File

<code>import java.io.File;</code>	
<code>public File(String nombre)</code>	Crea una instancia de la clase File , con el nombre de archivo completo como parámetro.
<code>public boolean exists()</code>	Devuelve true si el archivo especificado por el objeto existe, o false en caso contrario.
<code>public String getAbsolutePath()</code>	Devuelve el nombre de ruta completa del archivo especificado por el objeto.
<code>public String getName()</code>	Devuelve el nombre completo del archivo o directorio.
<code>public String getParent()</code>	Devuelve el nombre de ruta completa del padre del objeto.
<code>public boolean isDirectory()</code>	Devuelve true si este archivo existe y es un directorio, o false en caso contrario.
<code>public long length()</code>	Devuelve la longitud del archivo en bytes o cero si el archivo no existe.
<code>public String[] list()</code>	Devuelve un arreglo de nombres de archivo en el directorio especificado por el objeto archivo.
<code>public final static String separator</code>	El campo de datos contiene el valor del carácter separador de ruta, el cual es específico para el sistema operativo en particular.

● FileDialog

<code>import java.awt.FileDialog; o import java.awt.*;</code>	
<code>public FileDialog(Frame padre, String título, int modo)</code>	Crea una ventana de GUI de diálogo de archivos. El padre es el marco que crea la ventana. El título es el título de la ventana. El modo es FileDialog.LOAD o FileDialog.SAVE .
<code>public String getFile()</code>	Devuelve el nombre del archivo que se seleccionó o null si no se seleccionó ninguno.

● **FileInputStream**

<code>import java.io.FileInputStream; o import java.io.*;</code>	
<code>public FileInputStream(String nombre)</code>	Crea un flujo de archivo de entrada de un archivo con el nombre especificado.
<code>public void close()</code>	Cierra el flujo.

Vea también la clase `java.io.DataInputStream`.

● **FileOutputStream**

<code>import java.io.FileOutputStream; o import java.io.*;</code>	
<code>public FileOutputStream (String nombre)</code>	Crea un flujo de archivo de salida a partir de un archivo con el nombre especificado.

Vea también `java.io.PrintStream`.

● **FlowLayout**

Java ofrece varias formas de colocar componentes en la pantalla. La más sencilla es la que implica el uso de `FlowLayout`, y es la que aparece en la mayoría de los programas de este libro. A continuación le presentaremos el código familiar que utilizamos. En este ejemplo agregamos un panel para dibujar y dos botones:

```
private void crearGUI() {
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    Container ventana = getContentPane();
    ventana.setLayout(new FlowLayout());

    panel = new JPanel();
    panel.setPreferredSize(new Dimension(300, 200));
    panel.setBackground(Color.white);
    ventana.add(panel);

    botón1 = new JButton("botón1");
    ventana.add(botón1);
    botón1.addActionListener(this);
```

```

botón2 = new JButton("botón2");
ventana.add(botón2);
botón2.addActionListener(this);
}

```

Creamos un contenedor llamado `ventana` y después establecemos el estilo de esquema de flujo. De ahí en adelante los elementos que agregamos se colocan de izquierda a derecha, hasta llenar una fila. Después se empieza una nueva fila debajo de la anterior. Si no se llena por completo una fila de componentes, éstos se centran.

Si experimenta cambiando el tamaño de cualquiera de nuestros programas de gráficos, verá que se modifica el esquema dependiendo de la anchura del marco. El método del esquema de flujo es fácil de usar, pero tiene sus desventajas. Por ejemplo, tal vez queramos aumentar el tamaño del panel de gráficos a medida que aumenta el tamaño del marco, pero el esquema de flujo no permite esto. Consulte la sección sobre `BorderLayout` en este apéndice para obtener detalles sobre un método alternativo.

● JFrame

Un marco es una ventana de GUI de nivel superior con un título y un borde. Se utiliza como la base para las ventanas en casi todos los programas que presentamos en este libro.

<code>import javax.swing.JFrame; o import javax.swing.*;</code>	
<code>public Container getContentPane()</code>	Devuelve el objeto contenedor de este marco, para usarlo como contenedor de componentes de GUI. Vea la clase <code>Container</code> .
<code>public void setSize(int anchura, int altura)</code>	Ajusta el tamaño del marco, de manera que tenga la anchura <code>anchura</code> y la altura <code>altura</code> .
<code>public void setVisible(true)</code>	Hace visible el marco (lo muestra en pantalla).

● Graphics

```
import java.awt.Graphics; o import java.awt.*;
```

Este grupo de métodos provee herramientas para mostrar objetos gráficos simples, por lo general en un panel.

Un objeto se dibuja en el color actual, el cual se puede modificar con el método `setColor`. Los colores disponibles son `black`, `blue`, `cyan`, `darkGray`, `gray`, `green`, `lightGray`, `magenta`, `orange`, `pink`, `red`, `white`, `yellow`. Por ejemplo, para cambiar el color a `red` tenemos que hacer la siguiente llamada:

```
setColor(Color.red);
```



*Graphics continúa*

Las coordenadas de los objetos se expresan como coordenadas de píxeles *x* y *y*:

x se mide a través de la pantalla, desde la izquierda;

y se mide hacia abajo de la pantalla, con 0 en la parte superior.

Muchos de los objetos gráficos se consideran como si se dibujaran dentro de un rectángulo invisible (circundante), cuyas coordenadas de píxeles de la esquina izquierda superior son *x*, *y*. La altura y anchura del rectángulo, medidas en píxeles, son los otros dos parámetros.

<code>public void draw3DRect(int x, int y, int anchura, int altura, boolean elevado)</code>	Dibuja un rectángulo resaltado en 3D. Si elevado es true , el rectángulo aparece elevado.
<code>public void drawArc(int x, int y, int anchura, int altura, int ánguloInicial, int ánguloArco)</code>	Dibuja un arco circular o elíptico. El arco se dibuja desde ánguloInicial hasta ánguloInicial + ánguloArco . Los ángulos se miden en grados. Un ángulo de 0 es la posición de las 3 en punto. Un ánguloArco positivo va en sentido contrario a las manecillas del reloj.
<code>public void drawLine(int x1, int y1, int x2, int y2)</code>	Dibuja una línea a partir de las coordenadas. x1 , y1 a x2 , y2 .
<code>public void drawOval(int x, int y, int anchura, int altura)</code>	Dibuja una elipse o un círculo (si la anchura y la altura son iguales).
<code>public void drawPolygon(int puntosX[], int puntosY[], int puntosN)</code>	Dibuja un polígono cerrado utilizando los pares de coordenadas en los dos arreglos. Se dibuja una línea del primer punto al último punto.
<code>public void drawRect(int x, int y, int anchura, int altura)</code>	Dibuja un rectángulo utilizando el color actual.
<code>public void drawRoundRect(int x, int y, int anchura, int altura, int anchuraArco, int alturaArco)</code>	Dibuja un rectángulo con esquinas redondas en el color actual. anchuraArco es el diámetro horizontal del arco en las cuatro esquinas y alturaArco es el diámetro vertical.
<code>public void drawString(String s, int x, int y)</code>	Dibuja una cadena de texto. La línea base del primer carácter está en las coordenadas <i>x</i> , <i>y</i> . Los caracteres están en la fuente y color actuales.
<code>public void fillArc(int x, int y, int anchura, int altura, int ánguloInicial, int ánguloArco)</code>	Dibuja una forma de pastel utilizando el color actual. Consulte drawArc para los significados de los parámetros.
<code>public void fillOval(int x, int y, int anchura, int altura)</code>	Dibuja una elipse o círculo relleno.
<code>public void fillPolygon(int puntosX[], int puntosY[], int puntosN)</code>	Llena un polígono con el color actual. Consulte drawPolygon para el significado de los parámetros.
<code>public void fillRect(int x, int y, int anchura, int altura)</code>	Llena el rectángulo con el color actual.

```
public void fillRoundRect(int x,
    int y, int anchura, int altura,
    int anchuraArco, int alturaArco)

public void setColor(Color c)
```

Llena un rectángulo con esquinas redondas y en el color actual. Consulte `drawRoundRect` para el significado de los parámetros.

Establece el color actual.
Los colores disponibles se listan al principio de esta tabla.

A continuación le mostramos código de ejemplo. Hay un programa completo bajo la descripción de la clase `JPanel`. Una vez creado un panel, es necesario llamar al método `getGraphics` para obtener un objeto `Graphics`. Después se pueden utilizar los métodos.

```
JPanel panel = new JPanel();
Graphics papel = panel.getGraphics();
papel.drawLine(0,0,100,120);
```

● ImageIcon

```
import javax.swing.ImageIcon; o import javax.swing.*;
```

Esta clase permite mostrar una imagen de un archivo en un componente de GUI, por lo general, un panel.

```
public ImageIcon
    (String nombrearchivo)
```

Constructor.

Crea un objeto `ImageIcon`.

El nombre de archivo es el nombre de archivo completo, el nombre de archivo abreviado (si el archivo se encuentra en la misma carpeta que el archivo de clase del programa) o un URL.

```
public paintIcon(Component c,
    Graphics g, int x, int y)
```

Muestra la imagen en el área de gráficos `g`. `x`, `y` son las coordenadas de la esquina superior izquierda. `c` es el componente al que se notifica cuando se muestra la imagen en pantalla.

Por lo general es `this`.

```
public int getIconWidth()
```

Devuelve la anchura de la imagen.

```
public int getIconHeight()
```

Devuelve la altura de la imagen.

Programa de ejemplo:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class DemoImageIcon extends JFrame
    implements ActionListener {

    private JButton botón;
    private JPanel panel;
    private ImageIcon miImagen;
```



*ImageIcon continua*

```

public static void main(String[] args) {
    DemoImageIcon marco = new DemoImageIcon();
    marco.setSize(400,300);
    marco.crearGUI();
    marco.setVisible(true);
}

private void crearGUI() {
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    Container ventana = getContentPane();
    ventana.setLayout(new FlowLayout());
    panel = new JPanel();
    panel.setPreferredSize(new Dimension(150, 150));
    panel.setBackground(Color.white);
    ventana.add(panel);
    botón = new JButton("Haga clic");
    ventana.add(botón);
    botón.addActionListener(this);
    miImagen = new ImageIcon("imagenprueba.jpg");
}

public void actionPerformed(ActionEvent event) {
    Graphics papel = panel.getGraphics();
    int x = 10, y = 10;
    miImagen.paintIcon(this, papel, x, y);
}
}

```

**Integer**

Está en el paquete `java.lang`. No se requiere instrucción `import`.

```
public static int
parseInt(String s)
```

Convierte una cadena de dígitos decimales en un `int`. Puede ir un signo negativo antes de los dígitos. Se crea una excepción `NumberFormatException` si la cadena no contiene caracteres válidos.

```
public static String
toString(int i)
```

Convierte el `int` en la cadena de texto equivalente.

Código de ejemplo:

```
int i = Integer.parseInt(s);
String s = Integer.toString(i);
```

● JLabel

Una etiqueta de GUI, que se muestra con texto fijo.

```
import javax.swing.JLabel; o import javax.swing.*;
```

<code>public JLabel(String etiqueta)</code>	Crea una nueva etiqueta.
<code>public void setFont(Font f)</code>	Establece el tipo y tamaño de fuente. Ejemplo: <code>etiqueta.setFont(new Font(null, Font.BOLD, 60));</code>
	Para el constructor de <code>Font</code> , el segundo parámetro es el estilo. Las opciones son <code>Font.BOLD</code> , <code>Font.ITALIC</code> y <code>Font.PLAIN</code> .
	El tercer parámetro es el tamaño de la fuente.

Código de ejemplo:

```
private JLabel etiqueta;
etiqueta = new JLabel("algo");
```

● JList

Una lista es una lista de GUI de cadenas de texto (figura A.5). El usuario puede hacer clic en un elemento para seleccionarlo y esto produce un evento. El programa puede determinar el índice del elemento en el que se hizo clic (los valores de los índices empiezan en 0). El programa también puede determinar en qué texto se hizo clic.

Detrás de una lista de GUI hay una estructura de datos conocida como modelo de lista, la cual contiene la información a mostrar en pantalla. Los elementos se agregan a esta estructura.

<code>public JList(ListModel modelolista)</code>	Crea una nueva lista a partir de un modelo de lista.
<code>public void addListSelectionListener(Object objeto)</code>	Registra un objeto manejador de eventos.
<code>public int getSelectedIndex()</code>	Devuelve el índice del elemento seleccionado.
<code>public Object getSelectedValue()</code>	Devuelve el objeto seleccionado.

A continuación se muestra un programa de ejemplo. La lista se agrega a un objeto `JScrollPane`, lo cual significa que se muestran barras de desplazamiento si los datos son demasiado extensos para caber en el área de visualización disponible.

 *JList* continua

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;

public class DemoList extends JFrame implements ListSelectionListener {

    private JList lista;
    private DefaultListModel modeloLista;
    private JTextField campoTexto;

    public static void main(String[] args) {
        DemoList marco = new DemoList();
        marco.setSize(400,300);
        marco.crearGUI();
        marco.setVisible(true);
    }

    private void crearGUI() {
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        Container ventana = getContentPane();
        ventana.setLayout(new FlowLayout());

        modeloLista = new DefaultListModel();

        modeloLista.addElement("Mike");
        modeloLista.addElement("Maggie");
        modeloLista.addElement("Matthew");
        modeloLista.addElement("Eleanor");

        lista = new JList(modeloLista);
        lista.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
        lista.addListSelectionListener(this);
        ventana.add(new JScrollPane(lista));

        campoTexto = new JTextField(18);
        ventana.add(campoTexto);
    }

    public void valueChanged(ListSelectionEvent event) {
        int indice = lista.getSelectedIndex();
        String elemento = (String) lista.getSelectedValue();
        campoTexto.setText("elemento " + Integer.toString(indice)
            + " - " + elemento + " seleccionado");
    }
}
```

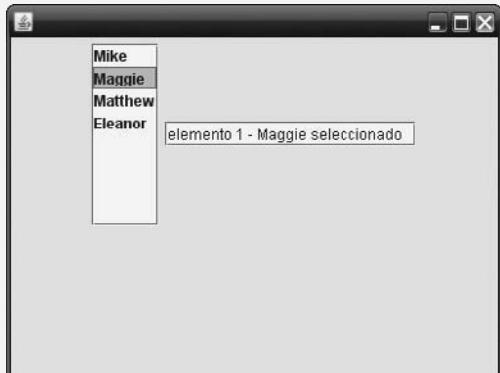


Figura A.5 Una lista.

● Math

Está en el paquete `java.lang`. No se requiere instrucción `import`.

Esta clase provee métodos matemáticos y dos constantes.

Todos los ángulos se expresan en radianes.

<code>public final static double E</code>	Valor de e.
<code>public static int abs(int a)</code>	Valor de π .
<code>public static int abs(int a)</code>	
<code>public static double abs(double a)</code>	
<code>public static double acos(double a)</code>	
<code>public static double asin(double a)</code>	
<code>public static double atan(double a)</code>	
<code>public static double ceil(double a)</code>	Devuelve el entero más cercano (como un valor <code>double</code>) que es mayor o igual a <code>a</code> .
<code>public static double cos(double a)</code>	e^a .
<code>public static double exp(double a)</code>	
<code>public static double floor(double a)</code>	Devuelve el entero más cercano (como valor <code>double</code>) que sea menor o igual a <code>a</code> .
<code>public static double log(double a)</code>	Logaritmo natural de base e.
<code>public static int max(int a, int b)</code>	Devuelve el valor más grande.
<code>public static double max(double a, double b)</code>	
<code>public static int min(int a, int b)</code>	Devuelve el valor más chico.



*Math continúa*

<code>public static double min(double a, double b)</code>	a^b .
<code>public static double pow(double a, double b)</code>	Devuelve un número seudoaleatorio en el rango de 0.0 hasta (sin incluir) 1.0.
<code>public static synchronized double random()</code>	Vea también la clase <code>java.util.Random</code> .
<code>public static double rint(double a)</code>	Devuelve el valor <code>double</code> que esté más cerca en valor a <code>a</code> y sea igual a un entero.
<code>public static double sin(double a)</code>	Seno de <code>a</code> .
<code>public static double sqrt(double a)</code>	Raíz cuadrada de <code>a</code> .
<code>public static double tan(double a)</code>	Tangente de <code>a</code> .
<code>public static int round(double a)</code>	Devuelve el valor más cercano a <code>a</code> .

Código de ejemplo:

```
y = Math.sqrt(x);
```

**Menu**

Esta clase cuenta con soporte para menús de GUI.

A continuación le mostramos un programa de ejemplo (vea la figura A.6). Se crea un objeto `JMenuBar`. A éste se agregan dos objetos `JMenu`. A su vez, se agregan elementos `JMenuItem` a estos menús. El conjunto completo de menús se agrega al panel mediante `setJMenuBar`.

```
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

public class DemoMenú extends JFrame implements ActionListener {

    JMenuBar barraMenúCompleta;
    JMenu menúArchivo, menúEdición;
    JMenuItem elementoAbrir, elementoGuardar, elementoCopiar, elementoPegar;
    JButton botónSalir;
    JTextField campoTexto;

    public static void main(String args[]) {
        DemoMenú marco = new DemoMenú();
        marco.setSize(250, 200);
        marco.clearRect();
        marco.setVisible(true);
    }
}
```

```
public void crearGUI() {  
    setDefaultCloseOperation(EXIT_ON_CLOSE);  
    Container ventana = getContentPane();  
    ventana.setLayout(new FlowLayout());  
  
    barraMenúCompleta = new JMenuBar();  
    setJMenuBar(barraMenúCompleta);  
  
    // menú Archivo con Abrir, Guardar  
    menúArchivo = new JMenu("Archivo");  
  
    elementoAbrir = new JMenuItem("Abrir");  
    menúArchivo.add(elementoAbrir);  
    elementoAbrir.addActionListener(this);  
  
    elementoGuardar = new JMenuItem("Guardar");  
    menúArchivo.add(elementoGuardar);  
    elementoGuardar.addActionListener(this);  
  
    barraMenúCompleta.add(menúArchivo);  
  
    // menú Edición con Copiar, Pegar  
    menúEdición = new JMenu("Edición");  
  
    elementoCopiar = new JMenuItem("Copiar");  
    menúEdición.add(elementoCopiar);  
    elementoCopiar.addActionListener(this);  
  
    elementoPegar = new JMenuItem("Pegar");  
    menúEdición.add(elementoPegar);  
    elementoPegar.addActionListener(this);  
    barraMenúCompleta.add(menúEdición );  
  
    botónSalir = new JButton("Salir");  
    ventana.add(botónSalir);  
    botónSalir.addActionListener(this);  
  
    campoTexto = new JTextField(10);  
    ventana.add(campoTexto);  
}  
  
public void actionPerformed(ActionEvent e) {  
    if (e.getSource() == elementoAbrir) {  
        campoTexto.setText("Seleccionó Abrir");  
    }  
    if (e.getSource() == elementoGuardar) {  
        campoTexto.setText("Seleccionó Guardar");  
    }  
    if (e.getSource() == elementoCopiar) {  
        campoTexto.setText("Seleccionó Copiar");  
    }  
}
```



*Menú continúa*

```

        if (e.getSource() == elementoPegar) {
            campoTexto.setText("Seleccionó Pegar");
        }
        if(e.getSource() == botónSalir) {
            System.exit(0);
        }
    }
}

```

**Figura A.6** Un menú.

● JOptionPane

Un panel de opción es un cuadro de GUI que un programa puede mostrar con un mensaje en su interior. Se necesita una instrucción `import` de la siguiente manera:

```
import javax.swing.JOptionPane;
```

Una llamada útil sería la siguiente:

```
JOptionPane.showMessageDialog(null, "aquí va su mensaje");
```

Esta instrucción crea la pantalla que se muestra en la figura A.7.

**Figura A.7** Panel de opción que se utiliza para pedir al usuario que confirme una opción.

El programa espera hasta que se hace clic en el botón **Aceptar**. Ésta es una llamada a un método estático de la clase **JOptionPane**.

Este componente también se puede utilizar para obtener información del usuario, como se muestra en este ejemplo:

```
String nombre = JOptionPane.showInputDialog("escriba su nombre");
```

La instrucción anterior crea la pantalla que se muestra en la figura A.8, en donde se invita al usuario a que escriba su nombre como entrada.

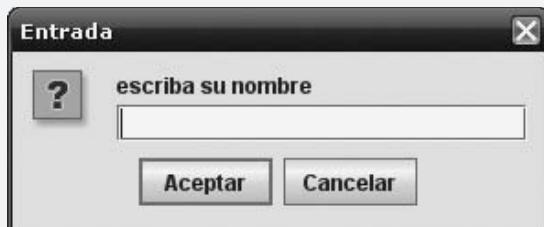


Figura A.8 Panel de opción que se utiliza para obtener datos del usuario.

● JPanel

Un área de dibujo. Se pueden dibujar objetos gráficos simples en un panel utilizando los métodos de la clase **Graphics**.

```
import javax.swing.JPanel; o import javax.swing.*;
```

public JPanel()	Crea un nuevo panel.
public Graphics getGraphics()	Devuelve el contexto de gráficos.
public int getHeight()	Devuelve la altura del panel en píxeles.
public int getWidth()	Devuelve la anchura del panel en píxeles.
public void setBackground(Color color)	Establece el color de fondo.
public void setPreferredSize(Dimension tamañoPreferido)	Establece el tamaño preferido. Por ejemplo: <code>panel.setPreferredSize(new Dimension(300, 200));</code>



*JPanel continua*

Programa de ejemplo (vea la figura A.9):

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class DemoPanel extends JFrame implements ActionListener {

    private JButton botón;
    private JPanel panel;

    public static void main(String[] args) {
        DemoPanel marco = new DemoPanel();
        marco.setSize(400,300);
        marco.createGUI();
        marco.setVisible(true);
    }

    private void createGUI() {
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        Container ventana = getContentPane();
        ventana.setLayout(new FlowLayout());

        panel = new JPanel();
        panel.setPreferredSize(new Dimension(100, 100));
        panel.setBackground(Color.white);
        ventana.add(panel);

        botón = new JButton("Haga clic");
        ventana.add(botón);
        botón.addActionListener(this);
    }

    public void actionPerformed(ActionEvent event) {
        Graphics papel = panel.getGraphics();
        papel.drawLine(0,0,111,113);
    }
}
```

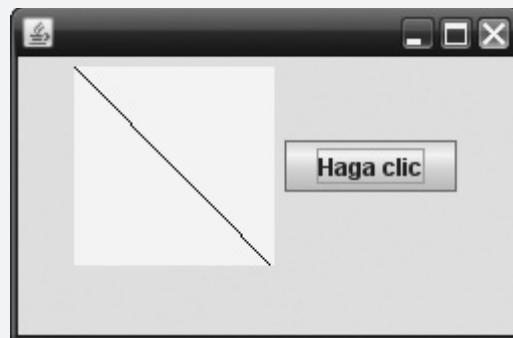


Figura A.9 Salida de DemoPanel. Se dibuja una línea en un panel.

Clícs del ratón

Un programa puede manejar los eventos de clic del ratón dentro de un panel. He aquí un programa de ejemplo que detecta un clic de ratón, dibuja un pequeño círculo en donde se hizo clic y muestra las coordenadas *x-y* del clic (vea la figura A.10). Debe proveer todos los métodos para manejar eventos de ratón, aun cuando no los utilice.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class DemoPanelConRatón extends JFrame implements MouseListener {

    private JPanel panel;
    private JTextField campoTexto;

    public static void main(String[] args) {
        DemoPanelConRatón marco = new DemoPanelConRatón();
        marco.setSize(400,300);
        marco.crearGUI();
        marco.setVisible(true);
    }

    private void crearGUI() {
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        Container ventana = getContentPane();
        ventana.setLayout(new FlowLayout());

        panel = new JPanel();
        panel.setPreferredSize(new Dimension(100, 100));
        panel.setBackground(Color.white);
        ventana.add(panel);
        panel.addMouseListener(this);

        campoTexto = new JTextField(10);
        ventana.add(campoTexto);
    }

    public void mouseClicked(MouseEvent evento) {
        int x = evento.getX();
        int y = evento.getY();
        Graphics papel = panel.getGraphics();
        papel.drawOval(x, y, 5, 5);
        campoTexto.setText(" x = " + Integer.toString(x)
            + "      y = " + Integer.toString(y));
    }

    public void mouseReleased(MouseEvent evento) {
    }
}
```





JPanel continúa

```
public void mousePressed(MouseEvent evento) {  
}  
  
public void mouseExited(MouseEvent evento) {  
}  
  
public void mouseEntered(MouseEvent evento) {  
}  
}  
}
```

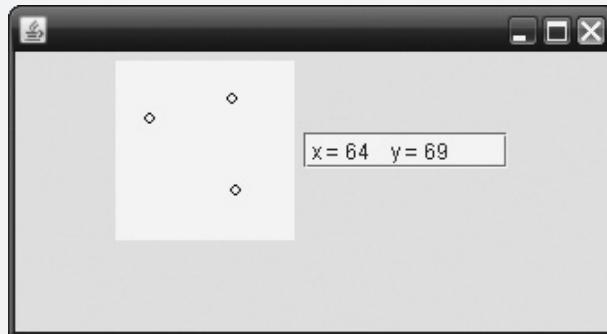


Figura A.10 Cómo manejar los clics del ratón en un panel.

Movimientos del ratón

Un programa puede manejar los eventos de movimiento del ratón. A continuación veremos un programa de ejemplo que detecta un movimiento del ratón, dibuja una línea desde la posición anterior hasta la nueva posición y muestra las coordenadas *x-y* del ratón (vea la figura A.11). Debe proveer los métodos de manejo de eventos y de movimiento del ratón, aun cuando no los utilice todos.

```
import java.awt.*;  
import java.awt.event.*;  
import javax.swing.*;  
  
public class Garabatos extends JFrame implements MouseMotionListener {  
  
    private JTextField campoTexto;  
    private JPanel panel;  
  
    int xAnt = 50, yAnt = 50;  
  
    public static void main(String[] args) {  
        Garabatos marco = new Garabatos();  
        marco.setSize(400,300);  
        marco.crearGUI();  
        marco.setVisible(true);  
    }  
}
```

```
private void crearGUI() {
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    Container ventana = getContentPane();
    ventana.setLayout(new FlowLayout());

    panel = new JPanel();
    panel.setPreferredSize(new Dimension(100, 100));
    panel.setBackground(Color.white);
    ventana.add(panel);
    panel.addMouseListener(this);

    campoTexto = new JTextField(8);
    ventana.add(campoTexto);
}

public void mouseDragged(MouseEvent evento) {
    int xNueva = evento.getX();
    int yNueva = evento.getY();
    Graphics papel = panel.getGraphics();
    papel.drawLine(xAnt, yAnt, xNueva, yNueva);

    xAnt = xNueva;
    yAnt = yNueva;

    campoTexto.setText("x = " + Integer.toString(evento.getX())
        + " y = " + Integer.toString(evento.getY()));
}

public void mouseMoved(MouseEvent evento) {
    int xNueva = evento.getX();
    int yNueva = evento.getY();
    xAnt = xNueva;
    yAnt = yNueva;
}
}
```

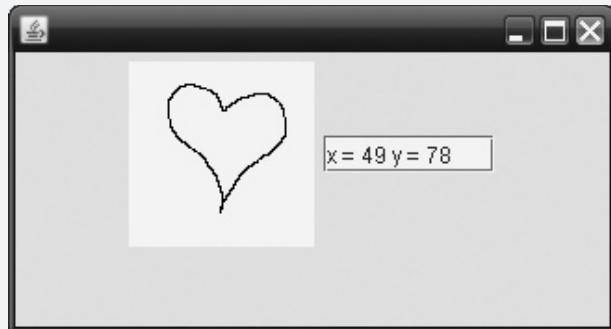


Figura A.11 Cómo manejar los eventos de movimiento del ratón en un panel.

● PrintStream

<code>import java.io.PrintStream; o import java.io;</code>	
<code>public PrintStream(OutputStream sal)</code>	Crea una instancia de un flujo de impresión que envía su salida al flujo de salida <code>sal</code> .
<code>public void close()</code>	Cierra el flujo de impresión.
<code>public void flush</code>	Vacia este flujo de impresión. La salida que esté en el búfer se escribe en el flujo.
<code>public void print(elemento)</code>	Imprime el elemento. Puede ser <code>char, double, float, int, long o String</code> .
<code>public void println(elemento)</code>	Imprime el elemento, seguido de un carácter de nueva línea.

● JRadioButton

Un grupo de botones de opción de GUI permite al usuario seleccionar una (pero sólo una) opción de las alternativas. Esto es similar a los botones de preselección en los primeros automóviles; sólo se podía seleccionar una estación de radio.

Al crear un botón de opción es necesario proporcionar la leyenda como parámetro. El segundo parámetro indica si debe estar seleccionado inicialmente o no.

Los botones de opción se deben agregar a un objeto `ButtonGroup` (no a un objeto `JButtonGroup`) para asegurar que sólo se seleccione uno de ellos en un momento dado.

Al cambiar una selección se crea un evento.

<code>public JRadioButton(String s, boolean seleccionado)</code>	Crea un botón de opción con la etiqueta <code>s</code> . El valor <code>boolean</code> determina si el botón está seleccionado inicialmente o no.
<code>public boolean isSelected()</code>	Devuelve <code>true</code> si el botón está seleccionado.

Programa de ejemplo (vea la figura A.12):

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class DemoJRadioButton extends JFrame implements ItemListener {

    private JRadioButton rojo, amarillo, azul;
    private ButtonGroup grupoBotones; // nota: sin J
    private JTextField campoTexto;
```

```
public static void main(String[] args) {
    DemoJRadioButton marco = new DemoJRadioButton();
    marco.setSize(400,300);
    marco.crearGUI();
    marco.setVisible(true);
}

private void crearGUI() {
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    Container ventana = getContentPane();
    ventana.setLayout(new FlowLayout());

    grupoBotones = new ButtonGroup();

    rojo = new JRadioButton("rojo", true);
    grupoBotones.add(rojo);
    ventana.add(rojo);
    rojo.addItemListener(this);

    amarillo = new JRadioButton("amarillo", false);
    grupoBotones.add(amarillo);
    ventana.add(amarillo);
    amarillo.addItemListener(this);
    azul = new JRadioButton("azul", false);
    grupoBotones.add(azul);
    ventana.add(azul);
    azul.addItemListener(this);

    campoTexto = new JTextField(10);
    ventana.add(campoTexto);
}

public void itemStateChanged(ItemEvent evento) {
    if (evento.getSource() == rojo) {
        campoTexto.setText("seleccionó rojo");
    }
    if (evento.getSource() == amarillo) {
        campoTexto.setText("seleccionó amarillo");
    }
    if (evento.getSource() == azul) {
        campoTexto.setText("seleccionó azul");
    }
}
```

Si desea verificar el estado de un botón de opción sin tener que manejar el evento, puede hacer lo siguiente:

```
if (rojo.isSelected()) etc.
```





JRadioButton continúa



Figura A.12 Botones de opción.

Esto es útil si, por ejemplo, tenemos un grupo de botones de opción y otro botón (ordinario) para iniciar alguna acción.

● Random

```
import java.util.Random;
```

Estos métodos devuelven lo que son aparentemente números aleatorios. De hecho, sólo son seudoaleatorios, ya que uno se calcula a partir del siguiente. Para asegurar que parezcan aleatorios, el primer valor se deriva de una semilla, que por lo general se basa en la hora actual (la parte de la hora actual correspondiente a los milisegundos siempre parece ser aleatoria). Para dar de manera explícita un valor a la semilla (cuando se realizan pruebas, por ejemplo) se utiliza `setSeed`.

Vea también el método `random` en la biblioteca `java.lang.Math`.

<code>Random()</code>	Crea un objeto generador de números aleatorios.
<code>public nextDouble()</code>	Devuelve el siguiente número seudoaleatorio, distribuido de manera uniforme del 0.0 al 0.999...
<code>public nextInt()</code>	Devuelve el siguiente número seudoaleatorio, distribuido de manera uniforme en el rango de <code>int</code> .
<code>public nextInt(int n)</code>	Devuelve un entero aleatorio en el rango de 0 a $n - 1$.
<code>public void setSeed(long semilla)</code>	Establece la semilla del generador de números aleatorios.

Código de ejemplo:

```
Random aleatorio = new Random();
double d = aleatorio.nextDouble();
campoTexto.setText("aleatorio es " + Double.toString(d));
```

● JSlider

Un objeto de GUI tipo control deslizable que se puede utilizar para cambiar valores de una manera ordinaria.

<code>import javax.swing.JSlider; o import javax.swing.*;</code>	
<code>JSlider()</code>	Crea un control deslizable horizontal en el rango de 0 a 100 y con un valor inicial de 50.
<code>JSlider(int mín, int máx, int valor)</code>	Crea un control deslizable horizontal utilizando los valores <code>mín</code> , <code>máx</code> y <code>valor</code> especificados.
<code>JSlider(int orientación, int mín, int máx, int valor)</code>	Crea un control deslizable con la orientación definida y los valores mínimo, máximo e inicial especificados. La orientación puede ser <code>JSlider.VERTICAL</code> o <code>JSlider.HORIZONTAL</code> .
<code>public void addChangeListener(Object objeto)</code>	Registra el objeto para manejar eventos.
<code>public int getValue()</code>	Devuelve el valor <code>int</code> .

Programa de ejemplo (vea la figura A.13), con un control deslizable que va de 0 a 10:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;

public class DemoSlider extends JFrame implements ChangeListener{

    private JSlider slider;
    private JTextField campoTexto;

    public static void main(String[] args) {
        DemoSlider marco = new DemoSlider();
        marco.setSize(400,300);
        marco.crearGUI();
        marco.setVisible(true);
    }

    private void crearGUI() {
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        Container ventana = getContentPane();
        ventana.setLayout(new FlowLayout());

        slider = new JSlider(JSlider.HORIZONTAL, 0, 10, 5);
        slider.addChangeListener(this);
        ventana.add(slider);
    }
}
```



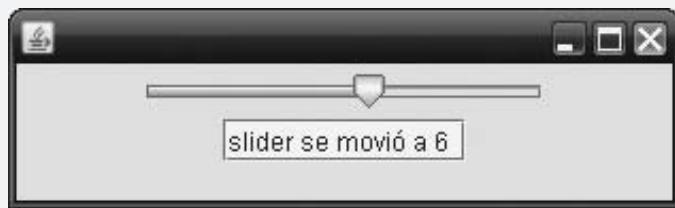
*JSlider continúa*

```

        campoTexto = new JTextField(10);
        ventana.add(campoTexto);
    }

    public void stateChanged(ChangeEvent e) {
        if (e.getSource() == slider) {
            campoTexto.setText("slider se movió a " +
                slider.getValue());
        }
    }
}

```

**Figura A.13** Un control deslizable y un campo de texto.

● String

Está en el paquete `java.lang`. No se requiere instrucción `import` ya que este paquete se importa de manera automática.

Cabe mencionar que se puede utilizar el operador `+` para concatenar (unir) cadenas.

<code>public char charAt(int índice)</code>	Devuelve el carácter en la posición especificada.
<code>public int compareTo(String cadena)</code>	Devuelve 0 si las cadenas son iguales. Devuelve un valor negativo si el objeto cadena va antes que el parámetro. Devuelve un valor positivo si el objeto cadena va después del parámetro.
<code>public boolean endsWith(String subcadena)</code>	Devuelve <code>true</code> si una cadena termina con una subcadena específica.
<code>public boolean equals(Object objeto)</code>	Devuelve <code>true</code> si la cadena tiene el mismo valor que el parámetro.
<code>public boolean equalsIgnoreCase(String cadena)</code>	Devuelve <code>true</code> si la cadena es igual que el parámetro; ignora el uso de mayúsculas/minúsculas.

<code>public int indexOf(String cadena, int índice)</code>	Devuelve el índice de la primera ubicación de la subcadena en la cadena, empezando desde la posición del segundo parámetro. Devuelve –1 si la subcadena no está presente.
<code>public int lastIndexOf(String cadena)</code>	Devuelve el índice de la ubicación de más a la derecha de la subcadena en la cadena. Devuelve –1 si la subcadena no está presente.
<code>public int length()</code>	Devuelve la longitud de la cadena.
<code>public String replace(char carAnt, char carNuevo)</code>	Devuelve la cadena con todas las ocurrencias de carAnt sustituidas por carNuevo .
<code>public String substring(int índiceInicial, int índiceFinal)</code>	Devuelve una parte específica de una cadena. El primer parámetro es la posición inicial. El segundo parámetro es la posición 1 más que el último carácter a extraer (observe la ortografía).
<code>public String toLowerCase()</code>	Devuelve la cadena convertida a minúsculas.
<code>public String toUpperCase()</code>	Devuelve la cadena convertida a mayúsculas.
<code>public String trim()</code>	Devuelve la cadena, pero elimina el espacio en blanco de ambos extremos.
<code>public static String valueOf(param)</code>	Espacio en blanco significa caracteres de espacio, de nueva línea y tabuladores.
	Convierte el parámetro en una cadena. El parámetro puede ser de tipo Object , char[] , boolean , char , int , long , float , double .

Vea también la clase `java.util.StringTokenizer`.

La cadena de tabulador es “\t”.

La cadena de nueva línea es “\n”.

● StringTokenizer

<code>import java.util.StringTokenizer;</code>	
<code>public StringTokenizer(String s, String delim)</code>	Crea un objeto StringTokenizer para la cadena s . Los caracteres en el parámetro delim son los delimitadores para separar los tokens.
<code>public boolean hasMoreTokens()</code>	Devuelve true si hay más tokens en la cadena o false en caso contrario.
<code>public String nextToken()</code>	Devuelve el siguiente token de este objeto StringTokenizer .

● System

Está en el paquete `java.lang`. No se requiere instrucción `import` ya que este paquete se importa de manera automática.

Contiene varios métodos útiles y también tres flujos convenientes.

<code>public static long currentTimeMillis()</code>	Obtiene la hora actual en milisegundos. Un milisegundo es 1/1000 de un segundo. El tiempo se mide desde medianoche del 1 de enero de 1970 UTC.
<code>public static PrintStream err</code>	El flujo de error estándar. Este flujo ya está abierto y listo para aceptar datos de salida. Por convención, este flujo se utiliza para mostrar los mensajes de error que deben requerir la atención inmediata del usuario.
<code>public static InputStream in</code>	El flujo de entrada estándar. Este flujo ya está abierto y listo para proveer datos de entrada. Por lo general, se introducen desde el teclado.
<code>public static PrintStream out</code>	El flujo de salida estándar. Este flujo ya está abierto y listo para aceptar datos de salida.
<code>public static void exit(int estado)</code>	Termina la aplicación y cierra la ventana. El estado debe ser 0 para una salida normal.

● JTextArea

Un componente de GUI que contiene líneas de texto; por lo general se usa para mostrar datos en pantalla.

<code>import javax.swing.JTextArea;</code> o <code>import javax.swing;</code>	
<code>public JTextArea(int altura, int anchura)</code>	Crea una nueva área de texto de un tamaño definido por <code>altura</code> y <code>anchura</code> , que se miden en caracteres. Tenga en cuenta que el orden de estos parámetros es el opuesto a todas las demás llamadas en la biblioteca en las que se utilizan anchura y altura.
<code>public void append(String s)</code>	Adjunta la cadena <code>s</code> al final del texto actual.
<code>public String getText()</code>	Devuelve el texto completo en el área de texto.
<code>public void insert(String s, int pos)</code>	Inserta la cadena <code>s</code> en la posición <code>pos</code> .

<code>public void replaceRange(String s, int inicio, int fin)</code>	Reemplaza el texto de inicio a (fin – 1) con la cadena s .
<code>public void setTabSize(int tamaño)</code>	Establece el tamaño del tabulador a tamaño .
<code>public void setText(String s)</code>	Coloca la cadena s en el área de texto.

Programa de ejemplo (vea la figura A.14) para mostrar texto con tabuladores y nuevas líneas:

```

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class DemoTextArea extends JFrame implements ActionListener {

    private JButton botón;
    private JTextArea áreaTexto;

    public static void main(String[] args) {
        DemoTextArea marco = new DemoTextArea();
        marco.setSize(400,300);
        marco.crearGUI();
        marco.setVisible(true);
    }

    private void crearGUI() {
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        Container ventana = getContentPane();
        ventana.setLayout(new FlowLayout());

        botón = new JButton("Haga clic");
        ventana.add(botón);
        botón.addActionListener(this);

        áreaTexto = new JTextArea(10, 10);
        ventana.add(áreaTexto);
    }

    public void actionPerformed(ActionEvent evento) {
        String nuevaLínea = "\r\n";
        String tab = "\t";
        áreaTexto.setTabSize(4);

        áreaTexto.setText(tab + "hola" + nuevaLínea);
        áreaTexto.append(tab + "java");
    }
}

```





JTextArea continúa



Figura A.14 Un área de texto.

Barras de desplazamiento

Si sabe cuánto espacio necesita, puede crear un área de texto de un tamaño apropiado. Si desea mostrar mucha información o si no sabe cuánta información habrá, es conveniente utilizar barras de desplazamiento a un lado del área de texto (vea la figura A.15).

El código para hacer esto es:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class ÁreaTextoDesplazable extends JFrame implements ActionListener {

    private JButton botón;
    private JTextArea áreaTexto;

    public static void main(String[] args) {
        ÁreaTextoDesplazable marco = new ÁreaTextoDesplazable();
        marco.setSize(400,300);
        marco.crearGUI();
        marco.setVisible(true);
    }
}
```

```
private void crearGUI() {  
    setDefaultCloseOperation(EXIT_ON_CLOSE);  
    Container ventana = getContentPane();  
    ventana.setLayout(new FlowLayout());  
  
    botón = new JButton("mostrar");  
    ventana.add(botón);  
    botón.addActionListener(this);  
  
    áreaTexto = new JTextArea(10, 10);  
    JScrollPane panelDesplazable = new JScrollPane(áreaTexto);  
    ventana.add(panelDesplazable);  
}  
  
public void actionPerformed(ActionEvent event) {  
    String nuevaLínea = "\r\n";  
    áreaTexto.setText("");  
    for (int conteo = 0; conteo < 100; conteo++) {  
        áreaTexto.append("línea " + conteo + nuevaLínea);  
    }  
}
```

Deben aparecer barras de desplazamiento en el extremo derecho y en la parte inferior del área de texto, según sea necesario. En este ejemplo sólo se necesita una barra para desplazarse vertical, aunque la figura muestra también una barra horizontal:

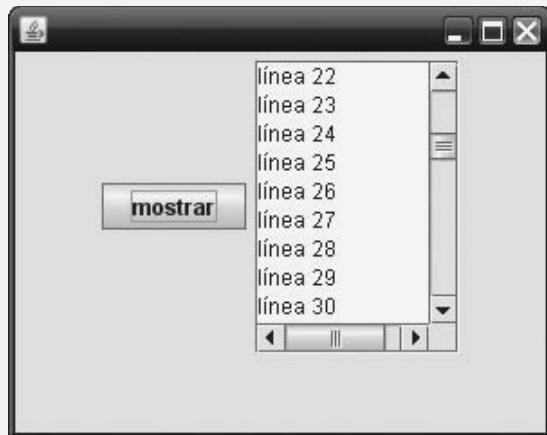


Figura A.15 Área de texto con una barra de desplazamiento.

● JTextField

Un componente de GUI que contiene una sola línea de texto, ya sea para entrada o salida.

```
import javax.swing.JTextField; o import javax.swing.*;
```

<code>public JTextField(int anchura)</code>	Crea un nuevo campo de texto, de anchura caracteres de ancho.
<code>public JTextField(String s, int anchura)</code>	Crea un nuevo campo de texto con texto inicial.
<code>public String getText()</code>	Devuelve el texto que está en el campo.
<code>public void setEditable(boolean b)</code>	Si b es true , el campo es editable; el usuario puede cambiar el valor en el campo de texto. Ésta es la opción predeterminada. Si b es false , el usuario no puede cambiar el valor en el campo de texto. Esto es útil para mostrar datos en pantalla.
<code>public void setFont(Font f)</code>	Establece el tipo y tamaño de fuente. Ejemplo: <code>campoTexto.setFont(new Font(null, Font.BOLD, 60));</code> En el constructor de Font , el segundo parámetro es el estilo. Las opciones son Font.BOLD , Font.ITALIC , Font.PLAIN . El tercer parámetro es el tamaño de la fuente.
<code>public void setText(String s)</code>	Coloca la cadena s en el campo de texto.

Programa de ejemplo, que copia lo que hay en un campo de texto en el otro al hacer clic en el botón (vea la figura A.16):

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class DemoTextField extends JFrame implements ActionListener {

    private JButton botón;
    private JTextField entrada, salida;

    public static void main(String[] args) {
        DemoTextField marco = new DemoTextField();
        marco.setSize(400,300);
        marco.crearGUI();
        marco.setVisible(true);
    }
}
```

```
private void crearGUI() {  
    setDefaultCloseOperation(EXIT_ON_CLOSE);  
    Container ventana = getContentPane();  
    ventana.setLayout(new FlowLayout());  
  
    entrada = new JTextField(8);  
    ventana.add(entrada);  
    botón = new JButton("haga clic");  
    ventana.add(botón);  
    botón.addActionListener(this);  
  
    salida = new JTextField(8);  
    ventana.add(salida);  
}  
  
public void actionPerformed(ActionEvent event) {  
    String texto;  
    texto = entrada.getText();  
    salida.setText(texto);  
}  
}
```



Figura A.16 Campos de texto.

● Thread

Está en el paquete `java.lang`. No se requiere instrucción `import`.

Esta clase provee los métodos para la programación multihilo.

<code>public boolean isAlive()</code>	Devuelve <code>true</code> si el hilo está en ejecución o bloqueado; <code>false</code> si el hilo es nuevo o está muerto.
<code>public final join()</code>	Espera hasta que haya terminado el hilo nombrado.
<code>public void run()</code>	Lo redefine un hilo. Este método constituye el código de un hilo que se inicia con el método <code>start</code> .
<code>public static void sleep(int m)</code>	El hilo se suspende durante <code>m</code> milisegundos. No hay garantía de que se despierte después del número requerido exacto de milisegundos. Este método debe proveer un manejador de excepciones para una excepción <code>InterruptedException</code> .
<code>public void start()</code>	Inicia la ejecución de un hilo. Un objeto se puede iniciar sólo una vez. Hay que volver a crearlo con <code>new</code> si necesitamos iniciarlo de nuevo.
<code>public void yield()</code>	Suspende este hilo temporalmente y permite que otro hilo obtenga tiempo del procesador.

● Timer

Un temporizador es un objeto que genera ciertos eventos de manera regular a intervalos preestablecidos. Se puede utilizar para controlar animaciones.

<code>public Timer(int retraso, ActionListener escucha)</code>	Crea un temporizador que notificará a su componente de escucha cada <code>retraso</code> milisegundos. Un milsegundo equivale a 1/1000 de un segundo.
<code>public void setDelay(int retraso)</code>	Establece el retraso del temporizador en milisegundos.
<code>Public void start()</code>	Inicia el temporizador. Empezará a crear eventos.
<code>public void stop()</code>	Detiene el temporizador. Ya no creará eventos.

He aquí un programa de ejemplo que utiliza la clase **Timer**. Muestra un panel de opción a intervalos de 10 segundos. La clase **Timer** está dentro del paquete **javax.swing**. No incluya la instrucción **import**:

```
import java.util.*;
```

ya que obtendrá acceso a otra clase llamada **Timer**, que es la incorrecta.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class DemoTimer extends JFrame implements ActionListener {

    Timer temporizador;

    public static void main(String[] args) {
        DemoTimer marco = new DemoTimer();
        marco.setSize(400,300);
        marco.crearGUI();
        marco.setVisible(true);
    }

    private void crearGUI() {
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        Container ventana = getContentPane();
        ventana.setLayout(new FlowLayout());

        temporizador = new Timer(10000, this);
        temporizador.start();
    }

    public void actionPerformed(ActionEvent event) {
        JOptionPane.showMessageDialog(null, "tic");
    }
}
```

APÉNDICE **B**



El Abstract Window Toolkit

● Introducción

El Abstract Window Toolkit (AWT) es un conjunto de clases de GUI que provee herramientas similares a Swing. A continuación veremos las generalidades del AWT.

● Swing y AWT

Cuando surgió Java a principios de la década de 1990, la estrategia que seguían los diseñadores de sus clases de GUI era utilizar partes de los componentes nativos de GUI. Así, un botón de Java en Microsoft Windows se veía como un botón de Microsoft Windows, y un botón de Java en una Apple Mac se veía como un botón de Apple Mac. Aunque esto funcionaba y era impresionante en su tiempo, tenía sus desventajas:

- Al migrar un programa de Java a otra plataforma cambiaba su apariencia visual.
- Como no todas las plataformas proporcionaban el mismo conjunto de componentes de GUI, el AWT tuvo que ser el mínimo común denominador. No contaba con muchos componentes.

Para solucionar estos problemas, Sun decidió escribir el conjunto de componentes Swing. Este conjunto no usa ninguno de los componentes nativos del sistema operativo y está escrito totalmente en Java. Los beneficios son:

- Los componentes se ven igual en cualquier plataforma. De hecho, Swing nos permite especificar que los componentes tengan una apariencia visual nativa o una apariencia visual de Swing. En este libro utilizamos esta última opción.
- Los diseñadores de los componentes pueden ir más allá del mínimo común denominador y crear un conjunto completo de componentes. Por ejemplo, Swing tiene componentes que proveen una

vista de árbol en la que se puede hacer clic (el estilo utilizado en el Explorador de Windows) y un componente tabla, que se asemeja a una hoja de cálculo. Como son componentes complejos, no los incluimos en este libro.

He aquí las principales clases del AWT. Cabe mencionar que su nombre no empieza con “J”.

- **Label**, **TextField** y **TextArea** con barras de desplazamiento integradas.
- **Checkbox** y **CheckboxGroup** [conocidos como botones de opción (radio buttons) en Swing].
- **Panel** para contener componentes y **Canvas** para los gráficos.
- Cuadros **Choice** [cuadros combinados (combo box) en Swing] y **List**.
- Un componente **Scrollbar** independiente [control deslizable (slider) en Swing], **Menu** y **FileDialog** [selector de archivos (file chooser) en Swing]. El AWT no provee cuadros de diálogo **JOptionPane**.

La lista anterior muestra la mayoría de los componentes del AWT. En Swing hay muchas más clases. Incluso en las clases similares que se muestran en la lista anterior, Swing provee más herramientas a través de sus métodos.

● Conversión de Swing a AWT

Probablemente ya esté familiarizado con las aplicaciones Swing que vimos en este libro. Por ello le proporcionaremos los lineamientos para convertirlas a AWT. Tenga en cuenta que, como Swing tiene más componentes y herramientas, no siempre es posible lograr una conversión sencilla. Pero si utiliza componentes sencillos, a menudo se pueden convertir de uno en uno (si lo vemos por el otro lado, convertir una aplicación de AWT a Swing también es un proceso simple).

La figura B.1 muestra una conversión del programa **SumarCamposTexto** del capítulo 6. He aquí el código:

```
import java.awt.*;
import java.awt.event.*;

public class SumarCamposTextoAWT extends Frame // marco
    implements WindowListener, ActionListener {

    private TextField campoNúmero1, campoNúmero2, campoSuma;
    private Label etiquetaIgual;
    private Button botónSuma;
```

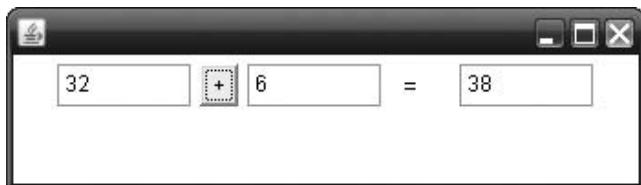


Figura B.1 Pantalla de **SumarCamposTextoAWT**.

```
public static void main(String[] args) {
    SumarCamposTextoAWT marco = new SumarCamposTextoAWT();
    marco.setSize(350, 100);
    marco.createGUI();
    marco.setVisible(true);
}

private void createGUI() {
    setLayout(new FlowLayout());

    campoNúmero1 = new TextField(7);
    add(campoNúmero1);

    botónSuma = new Button("+");
    add(botónSuma);
    botónSuma.addActionListener(this);

    campoNúmero2 = new TextField(7);
    add(campoNúmero2);

    etiquetaIgual = new Label(" = ");
    add(etiquetaIgual);

    campoSuma = new TextField(7);
    add(campoSuma);

    addWindowListener(this);
}

public void actionPerformed(ActionEvent event) {
    int número1 = Integer.parseInt(campoNúmero1.getText());
    int número2 = Integer.parseInt(campoNúmero2.getText());
    campoSuma.setText(Integer.toString(número1 + número2));
}

// código estándar para cerrar ventana, etc...

public void windowClosing(WindowEvent e) {
    System.exit(0);
}

// métodos de WindowListener vacíos
public void windowIconified(WindowEvent e) {
}
public void windowOpened(WindowEvent e) {
}
public void windowClosed(WindowEvent e) {
}
public void windowDeiconified(WindowEvent e) {
}
public void windowActivated(WindowEvent e) {
}
public void windowDeactivated(WindowEvent e) {
}

↑ // fin del código estándar de ventana
}
```

A continuación le mostramos algunos pasos para la conversión:

1. Elimine las instrucciones `import` de Swing de la parte superior del programa. Esto asegura que el compilador pueda detectar cualquier uso de Swing, aun si no se utiliza.
2. Elimine la “J” de todos los componentes. Así, `JFrame` se convierte en `Frame` y `JButton` se convierte en `Button`. No mezcle Swing y AWT en el mismo programa.
3. Busque un componente de AWT equivalente. Por ejemplo, tal vez tenga que usar un `Scrollbar` en vez de un `JSlider`.
4. Use la instrucción `implements WindowListener` y agregue el conjunto de métodos de ventana vacíos que se muestran en el listado anterior.
5. En el caso de las interfaces de usuario con esquema de flujo básico (flow layout) en donde los componentes se agregan en orden de izquierda a derecha, podemos agregarlas directamente al marco en vez de tener que crear un contenedor. Recuerde que:

```
add(botón1);
```

es abreviación de:

```
this.add(botón1);
```

en donde `this` representa al objeto actual. Es una instancia de la clase `Frame`.

6. Compile el programa. Es probable que tenga errores. Tenga en cuenta que hay variaciones en los nombres de los métodos y los tipos de eventos, por lo que tal vez necesite consultar la documentación del AWT en Web, en su IDE o en su sitio Web complementario.

● ¿Swing o AWT?

Si va a escribir una aplicación (como hicimos en los ejercicios del libro) use Swing, ya que provee muchas más herramientas y es ahora más popular que el AWT.

Si va a escribir applets que se van a ver en un navegador Web (vea el apéndice C), entonces la opción no es tan fácil. No todos los navegadores contienen las bibliotecas de Swing. Si su programa se va a utilizar en una organización con herramientas estándar, tal vez pueda garantizar lo que haya en cada computadora; por lo tanto, podría ser factible usar Swing. Pero si desea que el 100% de los usuarios de Internet puedan ejecutar su applet, use el AWT.

APÉNDICE C



Applets

Introducción

Este libro trata acerca de escribir “aplicaciones”. Se ejecutan bajo el control de su sistema operativo y el código de Java; los archivos de clase correspondientes se almacenan en su computadora. Los applets son distintos. El término significa programa pequeño. Los archivos de clase de los applets compilados se cargan en una computadora que actúa como servidor Web, en la misma carpeta en que podría almacenar sus páginas Web. Podemos especificar que una página Web incluya un vínculo a un applet. Cuando un usuario descargue dicha página, se incluirá con ella el código de las clases de Java y el applet se ejecutará en un área de la ventana del navegador Web.

Ejemplo de applet

A continuación veremos el proceso de crear y ejecutar un applet. Vamos a usar el programa **SumarCamposTexto** del capítulo 6. En el apéndice B proporcionamos una versión de este programa con el AWT; ésta es la versión que convertiremos en applet. La razón de esta elección es que los applets AWT trabajan con todos los navegadores, mientras que no todos los navegadores soportan Swing. La figura C.1 muestra la ejecución del applet dentro de un navegador Web. He aquí el código:

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

public class AppletSumarCamposTexto
    extends Applet implements ActionListener{

    private TextField campoNúmero1, campoNúmero2, campoSuma;
    private Label etiquetaIgual;
    private Button botónSuma;
```



Figura C.1 El applet `SumarCamposTexto` ejecutándose en el navegador de Microsoft.

```

public void init() {
    campoNúmero1 = new TextField(7);
    add(campoNúmero1);

    botónSuma = new Button("+");
    add(botónSuma);
    botónSuma.addActionListener(this);

    campoNúmero2 = new TextField(7);
    add(campoNúmero2);

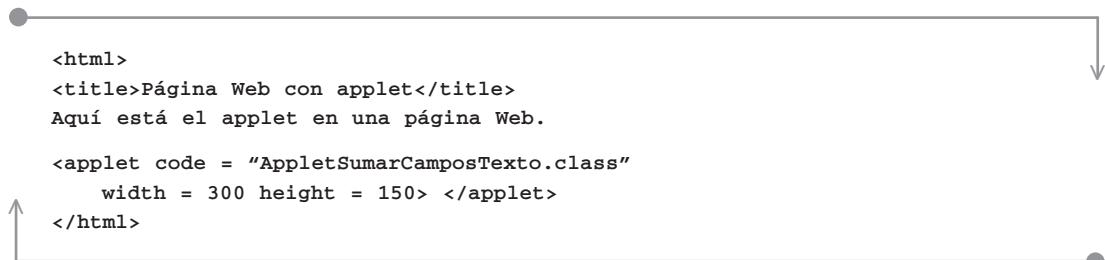
    etiquetaIgual = new Label(" = ");
    add(etiquetaIgual);

    campoSuma = new TextField(7);
    add(campoSuma);
}

public void actionPerformed(ActionEvent event) {
    int número1 = Integer.parseInt(campoNúmero1.getText());
    int número2 = Integer.parseInt(campoNúmero2.getText());
    campoSuma.setText(Integer.toString(número1 + número2));
}
}

```

El programa no se puede ejecutar por sí solo; hay que llamarlo desde una página Web. He aquí una página adecuada que contiene HTML, cuyo nombre es `demoapplet.html`:



La página Web utiliza una etiqueta especial de applet, la cual especifica el nombre del archivo de clase y el tamaño de la región de la pantalla del navegador en donde se ejecutará. Si utiliza un IDE, éste podría generar la página Web por usted. De no ser así, escríbala usted mismo pero modifique el nombre del applet y el tamaño, según sea apropiado.

Compile el programa como lo hace normalmente. Asegúrese de que el archivo de clase esté en la misma carpeta que el archivo que contiene la página Web que hace la llamada. Para ejecutar el applet en su computadora tiene dos opciones:

- Usar el programa Visor de Applets (Applet Viewer) que se incluye en su sistema de Java. Está diseñado para verificar su applet antes de usarlo dentro de una página Web.
- Use un navegador Web y abra la página que creó. Verá el applet ejecutarse en la ventana del navegador, como en la figura C.1.

Si tiene espacio Web en un servidor, cargue el archivo HTML y el archivo de clase compilado en su espacio Web. Así su applet estará disponible para cualquiera que desee usarlo. Si desea cargar también el código fuente de Java para que otros puedan utilizarlo, está bien. Pero no tiene que hacerlo.

Tenga en cuenta que cuando alguien descarga su applet, éste se ejecuta utilizando las bibliotecas de clase almacenadas dentro del navegador. Sun tiene un acuerdo que asegura que la mayoría de las bibliotecas de clases están disponibles en los navegadores; por ende se evita la descarga masiva de las bibliotecas de clases.

Diferencias de codificación de los applets

A continuación veremos las diferencias entre la programación de applets y la de aplicaciones. He aquí los puntos principales:

- Importar la biblioteca `Applet`:

```
import java.applet.Applet;
```

- Especificar que su programa extiende a `Applet`, como en:

```
public class AppletSumarCamposTexto
    extends Applet implements ActionListener {
```

- En vez de usar un método `main` para crear una nueva instancia del programa, usamos un método `init`. Este método es llamado por el navegador. Colocamos nuestro código de inicialización de GUI en este método.
- No tenemos que agregar código especial para terminar el programa. Esto se hace cuando el usuario se mueve a otra página Web.
- Los applets no tienen barra de menús, por lo que no se pueden utilizar estos elementos.
- Por cuestiones de seguridad, los applets no pueden leer o escribir archivos en la computadora en la que se ejecutan.

En el capítulo 26 hablamos sobre los aspectos de seguridad con más detalles.



Glosario

Aplicación –programa completo que se ejecuta de manera independiente.

Applet –programa pequeño (por lo general) que un navegador Web invoca desde una página Web. Algunas veces está escrito en Java, pero se puede escribir en otros lenguajes. El programa comúnmente interactúa con el usuario, muestra gráficos y animaciones, reproduce sonidos y videoclips. El término *applet* se deriva de las palabras en inglés “application” (aplicación) y “let”, que significa pequeño.

Clase abstracta –clase que actúa como plantilla para ciertas subclases. Provee todas las características comunes para las subclases. Una clase abstracta no se puede instanciar (no puede crear un objeto a partir de una clase abstracta).

Clase base – vea superclase.

Clase –unidad de programación en un lenguaje orientado a objetos como Java. Representa la abstracción de varios objetos similares (o idénticos). Describe los datos (variables) y métodos que contiene un objeto.

Código de bytes –lenguaje de bajo nivel en el que se compilan los programas de Java. Cuando se ejecuta un programa de Java, la Máquina Virtual de Java interpreta el código de bytes.

Extender – declarar una nueva clase que herede el comportamiento de otra clase.

Herencia –forma en que una nueva clase puede incorporar las características de una clase existente.

Instancia –objeto creado a partir de una clase.

Interfaz –característica de Java que permite al programador especificar la interfaz para una clase (la forma en que se va a utilizar), sin programar la clase en sí. Esto separa la especificación de lo que hará la clase de la forma en que lo va a hacer.

Java –lenguaje de programación de propósito general diseñado en un principio para construir aplicaciones destinadas a Internet.

Máquina Virtual de Java (JVM) –pieza de software que permite ejecutar Java en una máquina específica. La JVM interpreta el código de bytes producido por el compilador de Java.

Método – una de las acciones asociadas con un objeto. Un método tiene un nombre y puede tener parámetros (se le conoce también como *función*, *procedimiento* o *subrutina* en otros lenguajes de programación). El nombre *método* se deriva de la idea de tener un método para hacer algo.

Método de clase – vea método estático.

Método estático –método que pertenece a la clase como un todo y no a un objeto creado a partir de esa clase. Para invocar un método estático se utiliza el nombre de la clase como prefijo.

Objeto –componente de un programa en un lenguaje orientado a objetos. Un objeto incorpora datos (variables) y las acciones (métodos) asociadas con esos datos.

Palabra clave –nombre como **if**, **while**, **void** que forma parte del lenguaje Java y que el programador no puede utilizar como nombre.

Paquete –grupo de clases relacionadas, al cual se le asigna un nombre. El uso de paquetes evita el problema potencial de los nombres duplicados que podrían surgir en otras circunstancias, especialmente en piezas extensas de software.

Polimorfismo –característica de los lenguajes orientados a objetos, en la cual se selecciona en forma automática el método que va a actuar sobre un objeto.

Redefinición – proveer un método en una subclase que tenga el mismo nombre y parámetros que un método de una superclase. No se debe confundir con la sobrecarga.

Sobrecarga – proveer dos o más métodos (dentro de la misma clase) con el mismo nombre. Debe haber un número distinto de parámetros, los tipos de los parámetros pueden ser distintos, o se pueden dar ambos casos. El compilador de Java selecciona la versión del método correspondiente a los parámetros que se utilicen en el momento de invocar el método. No se debe confundir con la redefinición.

Subclase –clase que hereda variables y métodos de la clase actual.

Superclase – clase a partir de la cual la clase actual hereda variables y métodos. A menudo se le conoce como clase base.

Tipos integrados – los tipos **boolean**, **char**, **byte**, **short**, **int**, **long**, **float** y **double** que se proveen listos para usarse como parte del lenguaje Java. No son verdaderos objetos.

Variable a nivel de clase – vea variable de instancia.

Variable de clase – vea variable estática.

Variable de instancia –variable que se declara en la parte superior de una clase. Se hace una copia de esta variable para cada objeto creado a partir de la clase. No se debe confundir con una variable estática.

Variable estática –variable que se declara en la parte superior de una clase y pertenece a la clase como un todo. No se copia cuando se crea un objeto a partir de esa clase.

Widget –botón, barra de desplazamiento, campo de texto u objeto similar en una ventana que cuente con soporte para la interacción del usuario cuando éste haga clic con el ratón. Se deriva de las palabras en inglés “window” (ventana) y “gadget” (utensilio).

APÉNDICE **E**



Reglas para los nombres

En Java el nombre de una variable, una clase o un método se conoce como identificador. El programador selecciona los identificadores. Las reglas son:

- Un identificador puede tener la longitud que usted quiera.
- Un identificador consiste en letras (mayúsculas y minúsculas) y dígitos (0 al 9).
- Un identificador debe empezar con una letra.
- Un identificador no debe ser igual que una de las palabras clave de Java que se listan en el apéndice F.

Una letra se define como un carácter de la A a la Z, de la a a la z, un signo de dólares (\$) o un guión bajo (_). Por lo general, el signo de dólares y el guión bajo no se utilizan; los incluimos por motivos históricos. En la categoría de las letras también se incluyen las letras de muchos lenguajes del mundo.

Por convención:

- Los nombres de las clases empiezan con una letra mayúscula y continúan con minúscula, como **Graphics**.
- Los nombres de las variables y los métodos empiezan con letra minúscula, como **sumar**.
- Cuando un nombre está formado por una combinación de palabras, la segunda palabra y las subsiguientes empiezan con letra mayúscula, como **deseoQueEstésAquí**, **drawLine**.
- Los nombres de los paquetes empiezan con letra minúscula.

Las letras mayúsculas y minúsculas son válidas y distintas, por lo que **ratón** es distinto de **RATÓN** y de **Ratón**.

APÉNDICE

F



Palabras clave

A continuación le mostramos las palabras clave de Java. El programador no las puede usar como nombres para variables, métodos, clases o paquetes.

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

Las palabras clave **const** y **goto** son reservadas, aun cuando no se utilizan.

true y **false** parecen ser palabras clave, pero en realidad son literales **boolean**. De manera similar, aunque **null** parezca ser una palabra clave, técnicamente es la literal nula. Sin embargo, el programador no puede usar ninguna de estas palabras (por lo que podríamos considerarlas como palabras clave).



Reglas de alcance (visibilidad)

La visibilidad es el término que se utiliza para describir qué elementos pueden acceder (o hacer referencia) a qué otros elementos. En este apéndice presentamos varias versiones de las reglas de Java. Hemos incluido una variedad debido a que tal vez encuentre algunas de ellas más fáciles de comprender y de usar que otras.

Recuerde: las variables y los métodos se agrupan en clases; después las clases se agrupan en paquetes. Para describir un método o una variable utilizamos uno de los modificadores en la lista que se muestra a continuación. Este modificador describe el tipo de acceso permitido por el código en la misma clase o en otras clases. El acceso a un método significa el permiso de llamar al método. El acceso a una variable significa el permiso para usar su valor o modificarlo (pero en general, el acceso a las variables no es conveniente ya que viola el principio del ocultamiento de información).

Las reglas para acceder a los métodos o las variables son:

Modificador de método o variable	Significado
<code>public</code>	Accesible desde cualquier parte.
<code>protected</code>	Accesible desde: 1. Esta clase. 2. Cualquier subclase. 3. Cualquier clase en el mismo paquete.
predeterminado – sin modificador	Accesible desde: 1. Esta clase. 2. Cualquier clase en el mismo paquete.
<code>private</code>	Accesible sólo desde el interior de esta clase.

Esta información se muestra en un formato alternativo en la siguiente tabla. Por coincidencia, esto ilustra la uniformidad del esquema de acceso:

Modificador de método variable	¿Accesible desde la misma clase?	¿Accesible desde cualquier clase en el mismo paquete?	¿Accesible desde cualquier subclase?	¿Accesible desde cualquier parte?
public	sí	sí	sí	sí
protected	sí	sí	sí	no
predeterminado – ninguno	sí	sí	no	no
private	sí	no	no	no

El segundo tipo de acceso a las variables y los métodos es el que proporciona la herencia. La siguiente tabla describe las reglas:

Modificador de método o variable	¿Cualquier subclase en el mismo paquete lo puede heredar?	¿Cualquier subclase lo puede heredar?
public	sí	sí
protected	sí	sí
predeterminado – ninguno	sí	no
private	no	no

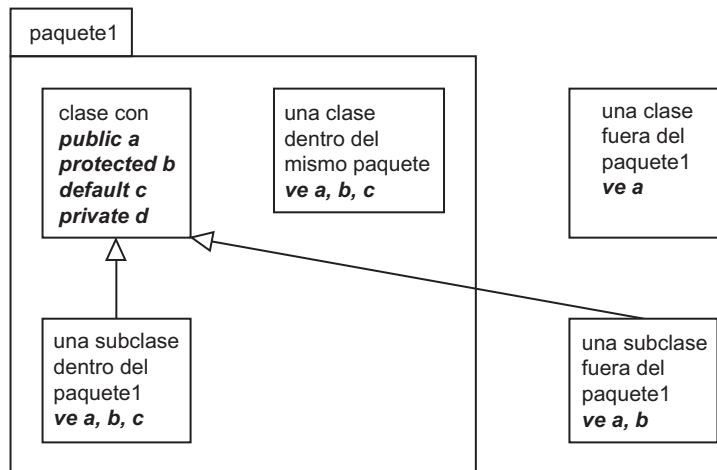
Para una clase hay menos opciones. Las reglas son:

Modificador de clase	Significado
public	Cualquier clase puede usar esta clase.
predeterminado – ninguno	Sólo las clases del mismo paquete pueden usar esta clase.

Los constructores siguen las mismas reglas que los métodos, pero vale la pena describirlas. Éstas son:

Modificador del constructor	Significado
<code>public</code>	Cualquier clase puede crear una instancia de esa clase mediante una llamada a este constructor.
<code>protected</code>	Las únicas clases que pueden crear una instancia de esta clase son: 1. Las subclases. 2. Las clases en el mismo paquete.
predeterminado – ninguno	Sólo las clases en el mismo paquete pueden crear una instancia de esta clase.
<code>private</code>	Ninguna otra clase puede llamar a este constructor (pero un método de clase dentro de esta clase puede llamar a este constructor para crear un objeto).

He aquí un diagrama de UML que muestra las reglas de visibilidad de Java.



El rectángulo grande denota un paquete. Su nombre es paquete1, como se muestra en la pestaña de la parte superior izquierda. Los demás rectángulos denotan clases. Tres están dentro del paquete y dos están fuera de él. Dentro del paquete hay una clase que declara los métodos/variables a, b, c y d. El diagrama muestra el tipo de acceso que tiene cada una de las demás clases sobre estos métodos/variables.

APÉNDICE

H



Bibliografía

● Java

En realidad no es necesario ningún otro libro más que el que está leyendo. Aquí está toda la información necesaria para escribir una variedad extensa de programas (y sin duda los de los ejercicios también). No olvide el apéndice sobre las bibliotecas de Java. Los libros que indicaremos a continuación le mostrarán otras perspectivas sobre Java.

The Java Programming Language, Ken Arnold y James Gosling, Addison-Wesley, varias ediciones. Escrito por los miembros del grupo de Sun que crearon Java. La mejor fuente de información sobre Java. Este libro describe el lenguaje en sí (y no las bibliotecas). Es muy comprensible, pero no para principiantes.

The Java Developers Almanac, Patrick Chan, Addison-Wesley, ediciones frecuentes. Una vez que comprenda el lenguaje Java, tendrá que entender lo que proporcionan las bibliotecas. Estos libros de referencia de dos volúmenes listan todas las clases y todos los miembros de éstas. Una referencia útil.

Cómo programar en Java, Paul Deitel y Harvey Deitel, Pearson Educación-Prentice Hall, 2008. Es muy completo y formidable. Útil como referencia.

Tal vez la mejor forma de obtener información sobre las bibliotecas sea utilizar su motor de búsqueda favorito. Por decir, si necesita información sobre la clase `JTextField`, puede escribir:

Sun Java 6 JTextField

UML

He aquí dos libros cortos y simples sobre UML:

UML Distilled, 3^a. edición, Martin Fowler con Kendall Scott, Addison-Wesley, Harlow, 2004.

Using UML: Software Engineering with Objects and Components, 2nd edition, Perdita Stevens, Addison-Wesley, Harlow, 2006.

Lenguajes de programación orientados a objetos

Java es uno de los más recientes lenguajes orientados a objetos. Todos comparten los conceptos de encapsulamiento, clases y herencia.

Smalltalk-80: the language, Adele Goldberg y David Robson, Addison-Wesley, Reading, MA, 1989. Smalltalk-80 es el Rolls-Royce de los lenguajes orientados a objetos. Es totalmente orientado a objetos; incluso las estructuras de control como la repetición y las instrucciones `if` son objetos. Al igual que Java, soporta la herencia simple. Al igual que Java, provee una biblioteca grande y completa que el programador utiliza y de la cual hereda para proveer herramientas como ventanas, gráficos y estructuras de datos. El mejor libro sobre Smalltalk-80.

Object-Oriented Software Construction, Bertrand Meyer, Prentice Hall, Nueva York, 2000.

El mejor libro sobre el lenguaje Eiffel. Los primeros capítulos son una maravillosa y clara exposición de los principios de la POO.

Desarrollo de software

Writing Solid Code: Microsoft's techniques for developing bug-free C programs, Steve Maguire, Microsoft Press, Redmond, WA, 1993.

Un libro de un programador de Microsoft que describe técnicas para depurar software. Hay muchos ejemplos en el lenguaje de programación C.

Software Engineering, Douglas Bell, Prentice Hall, Harlow, 2005.

La ingeniería de software es el término que se da a la tarea de desarrollar programas extensos. Este libro describe las metodologías para esta tarea; supone que el lector tiene conocimientos de programación.



Instalación y uso de Java

En este apéndice le proporcionamos información y consejo sobre cómo instalar Java en Microsoft Windows, aunque la mayor parte de los detalles sobre Eclipse también se aplican a GNU/Linux.

Hay tres posibilidades:

1. Usar un IDE (entorno de desarrollo integrado), como Eclipse.
2. Usar un editor de texto que esté configurado para Java.
3. Usar la línea de comandos. No es una opción práctica para principiantes, pero algunas veces se requiere para tareas más avanzadas.

Cada una de estas opciones requiere que primero descargue el software de Java de Sun. Enseguida mostraremos más detalles para cada opción.

● Descarga de Java

Puede descargar Java para Microsoft Windows y GNU/Linux sin costo desde el sitio Web de Sun:

java.sun.com

Para la Mac (OS X) vaya a:

developer.apple.com/java/

El proceso de descarga y los nombres de los archivos varían, pero puede seleccionar el vínculo **Downloads** y localizar la página relacionada con:

Java SE Development Kit 6

Después localice el JDK, que tendrá un nombre parecido a:

JDK 6 Update XX

Tal vez se incluya junto con otro software. Por simpleza, evite descargar un JDK incluido con NetBeans o Java EE.

Seleccione la versión apropiada para GNU/Linux o Microsoft Windows. En caso de que Sun cambie a la versión 7 y la versión 6 no esté disponible, seleccione la 7. Los programas de este libro trabajarán con la versión 7.

El archivo se llamará

`jdk-6u14-windows-i586.exe`

o algo por el estilo. Al descargar el archivo, podrá guardarlo en cualquier carpeta y borrarlo después de la instalación.

Para instalar en Microsoft Windows, ejecute el archivo. Es más simple aceptar todas las opciones predeterminadas. Tenga en cuenta que una opción le pregunta qué características desea instalar. Al hacer clic en **Next** se instalan todas, lo cual es la mejor opción.

Cabe mencionar que el programa que acaba de instalar no hace nada por sí solo. Necesita Eclipse o un editor de texto para crear y ejecutar programas.

● El IDE Eclipse

Hay varios IDE disponibles. Nosotros le recomendamos Eclipse. Éste es un programa de código fuente abierto establecido, escrito en Java. Además es gratuito.

Las ventajas de utilizar un IDE en vez de un editor de texto son:

- Tiene integrado el conocimiento del lenguaje Java y las bibliotecas, por lo que el IDE puede proveer sugerencias a medida que va escribiendo.
- Hay un depurador integrado que permite insertar puntos de interrupción y avanzar paso a paso por las instrucciones, como vimos en el capítulo 21.

La principal desventaja de un IDE es su complejidad para un principiante que desea escribir un programa pequeño, si se le compara con un editor de texto, que es más sencillo. En general, creemos que vale la pena invertir el tiempo en familiarizarse con un IDE.

Para usar Eclipse necesita instalarlo y después configurarlo.

Instalación de Eclipse

Asegúrese de haber descargado e instalado Java.

Primero, descargue Eclipse de

www.eclipse.org

Vaya a la página de descargas y localice

Eclipse IDE for Java Developers (para Windows)

Cabe mencionar que también hay versiones para GNU/Linux y Mac. Su nombre es similar a:

`eclipse-java-xxxx-win32.zip`

No descargue la versión de Java EE.

El archivo es de 90 Mbytes o más. Guárdelo en su disco duro. Podrá eliminarlo después de haber instalado el software.

Cómo descomprimir Eclipse

Los archivos de Eclipse están comprimidos (en un archivo grande), por lo que necesita descomprimirlas en una ubicación apropiada (como la carpeta **Archivos de programa**).

A continuación le mostraremos cómo descargar una versión de prueba del programa WinZip:

Vaya a

www.winzip.com

En la página de descargas hay una versión de prueba. Descárguela y guárde la en una carpeta; después ejecute el archivo. No necesita instalar la barra de herramientas de Google, en caso de que el sistema le haga esa pregunta.

Hay una pregunta relacionada con la instalación de una lista de elementos; instálelos todos.

Por último, seleccione el estilo “clásico” si el sistema le da la opción.

Ahora está listo para descomprimir Eclipse. Puede ejecutar WinZip y abrir el archivo de Eclipse o puede hacer clic con el botón derecho en el archivo y seleccionar WinZip.

Necesita especificar una ubicación para guardar los archivos descomprimidos. Puede ser cualquier ubicación, pero es conveniente usar:

C:\Archivos de programa

y después haga clic en **Extract**. El proceso de descompresión puede tardar varios minutos.

Ahora deberá tener una carpeta llamada **eclipse** dentro de la carpeta **Archivos de programa**.

Cómo ejecutar Eclipse

Dentro de la carpeta de Eclipse encontrará un programa llamado **eclipse.exe**. Haga doble clic en este programa para ejecutarlo. Es conveniente hacer clic con el botón derecho sobre el archivo y crear un acceso directo, el cual puede arrastrar a su directorio.

La primera vez que ejecute Eclipse, realizará ciertas tareas de preparación y se ejecutará con lentitud, por lo que debe ser paciente. En cierto momento aparecerá una ventana llamada **Workspace Launcher**. Debe proporcionar a Eclipse el nombre de una carpeta en la que desee almacenar todos sus programas de Java. La configuración predeterminada es:

C:\Documents and Settings\Propietario\workspace

Puede aceptar esta opción. O tal vez prefiera una carpeta a un nivel menos profundo en la estructura de archivos, como:

c:\JavaWorkspace

Puede elegir su carpeta. Seleccione la casilla **Do not ask again** (No preguntar de nuevo) y haga clic en **OK**.

Después de un tiempo aparecerá un panel de bienvenida. Como veremos más adelante, Eclipse muestra muchos paneles en la pantalla en forma simultánea; puede quitar los que no deseé si hace clic en la X en su esquina superior izquierda. Ahora puede quitar el panel de bienvenida.

La próxima vez que ejecute Eclipse no aparecerá la pregunta sobre el espacio de trabajo ni el panel de bienvenida.

Creación de un proyecto en Eclipse

Eclipse está destinado a la programación a gran escala, en donde los programas se crean a partir de muchos archivos de clases de Java separados. Tiene el concepto de un proyecto, que es una colección de archivos. Incluso si queremos crear un solo programa en un archivo, debemos crear un proyecto, el cual se almacena en nuestro espacio de trabajo en Eclipse.

Vamos a utilizar algunos de los menús desplegables en la parte superior de la pantalla. Utilizaremos esta notación:

File > New

significa “hacer clic en el menú **File** y seleccionar la opción **New**”.

Ahora de vuelta a la creación de nuestro proyecto. Si aparece algún proyecto existente (que es poco probable, ya que todavía no hemos creado uno) entonces seleccione:

File > Close All

Para iniciar la creación seleccione:

File > New > Java Project

A continuación se abrirá una ventana titulada **New Java Project**; necesita introducir el nombre de un proyecto. El nombre puede ser cualquier cosa que usted seleccione, pero es preferible un nombre significativo. Por ejemplo, imagine que va a escribir un programa de Java que necesita almacenar en:

HolaMundo.java

Un nombre apropiado para el proyecto podría ser:

Proyecto HolaMundo

Los nombres de los proyectos pueden incluir espacios.

Ignore el resto de las opciones en la pantalla y haga clic en **Finish**.

Un proyecto puede contener varios archivos (clases) de Java. Ahora necesitamos agregar una clase a nuestro proyecto, para lo cual debemos hacer clic en:

File > New > Class

A continuación se abrirá una ventana titulada **New Java Class**. Localice el indicador **Name:** y escriba el nombre de su clase (por ejemplo, **HolaMundo**). No agregue la terminación **.java** después del nombre.

Ignore todas las demás opciones y haga clic en **Finish**.

Ahora el proyecto aparecerá como se muestra en la figura I.1.

Tenga en cuenta lo siguiente:

- El panel **Package Explorer** muestra el espacio de trabajo, con un proyecto dentro de éste.
- El proyecto se almacena en una carpeta llamada **Proyecto HolaMundo**, la cual contiene una carpeta **src** (código fuente).
- Dentro del paquete predeterminado (**default package**) se encuentra **HolaMundo.java**, que sólo incluye unas cuantas líneas de Java. Puede eliminarlas si lo desea, para escribir (o pegar) nuevo código de Java para la clase **HolaMundo**.
- Si no puede ver el archivo de Java, necesita recorrer el proyecto; abra (haciendo doble clic) la carpeta **src** y **default package**.

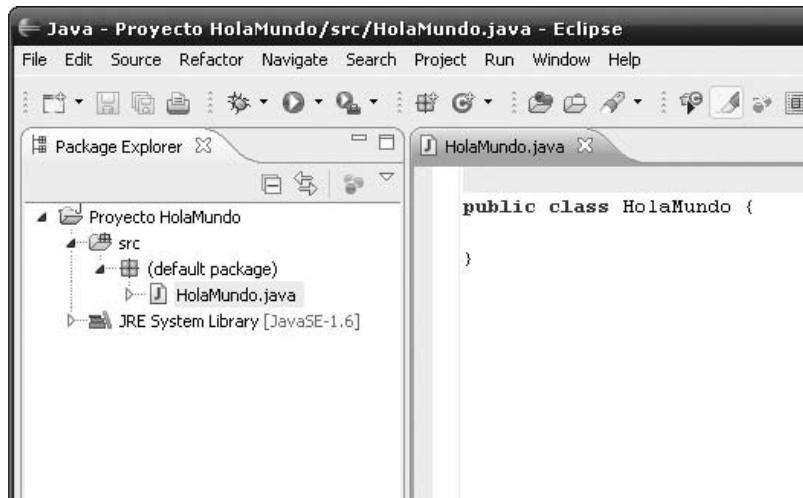


Figura I.1 Un proyecto de Java en Eclipse.

- Tenga en cuenta que es complicado cambiar el nombre de la clase en esta etapa; altere el código de Java para que coincida con el nombre elegido, en vez de hacer lo contrario.

Ejecución de un programa con Eclipse

Si creó la clase `HolaMundo` que vimos en la sección anterior, ahora le mostraremos un programa que puede escribir en ella. Tenga en cuenta que el nombre de la clase en la primera línea debe ser exactamente el nombre que dimos a Eclipse antes: con una H y M mayúsculas. En cuanto al resto, ¡no se preocupe por los detalles del lenguaje todavía!

```
public class HolaMundo{
    public static void main(String[] args){
        String s = " mun";                                // línea 3
        s = s + "do";
        System.out.println("hola" + s);
    }
}
```

Cabe mencionar que su programa se revisa y compila a medida que lo escribe. Los errores se muestran mediante un símbolo de advertencia en el margen; coloque el ratón sobre el símbolo para obtener una descripción del error. Cometa un error intencional al escribir el código anterior para que pueda ver esta característica en acción.

Para ejecutar el programa, seleccione:

Run > Run

La primera vez que ejecute un programa aparecerá la ventana **Save and launch** (Guardar y lanzar), que le preguntará si desea guardar los archivos. La opción más segura es **Select All** (Seleccionar todo) y después marcar la casilla **Always save resources before launching** (guardar siempre los recursos antes de lanzar).

El programa anterior debe mostrar el mensaje **hola mundo** en el panel de consola ubicado en la parte inferior de la pantalla.

Eclipse le puede ayudar a aplicar sangría automáticamente a su código mediante la opción **Source > Format**. Seleccione **File > Exit** para terminar su sesión.

Más adelante veremos cómo volver a abrir los proyectos.

Depuración

La depuración se describe en el capítulo 21. A continuación le mostraremos algunas técnicas específicas de Eclipse.

Para crear un punto de interrupción en una línea de código, haga clic con el botón derecho del ratón en el margen sombreado a la izquierda de la línea y seleccione:

Toggle breakpoint

Por ejemplo, podría colocar un punto de interrupción en la línea 3 del programa anterior (después podrá quitar el punto de interrupción mediante la misma opción **Toggle breakpoint**).

Para depurar el programa, no lo ejecute de la manera normal. Utilice la siguiente opción:

Run > Debug

La primera vez que depura un programa aparece una ventana titulada **Confirm Perspective Switch** (confirmar cambio de perspectiva). Marque la casilla **Remember my decision** (recordar mi decisión) y después haga clic en **Yes**.

A continuación aparecerá una pantalla de detalles de depuración y se resaltará la línea con el punto de interrupción. Esto indica que la línea está a punto de ejecutarse. Por ende, en la línea 3 anterior, la cadena **s** no se ha declarado todavía.

Para ejecutar la línea, seleccione **Run > Step Into**. Observe el panel **Variables** que muestra el valor actual de las variables. A continuación verá que se lista el valor “**mun**” para **s**. Al avanzar a la siguiente línea, la **s** cambia a “**mundo**”.

¡El depurador es bastante complejo! Éstas son otras herramientas que le podrían ser de utilidad:

- **Run > Step over** Si la línea a ejecutar contiene una llamada a un método, el método se ejecutará a la velocidad normal en vez de que usted tenga que avanzar paso a paso por cada línea manualmente; esto es útil cuando sabemos que el método no tiene errores.
- **Run > Resume** Continúa ejecutando el programa a velocidad normal; es útil para avanzar entre varios puntos de interrupción con rapidez.

Cómo volver a abrir un proyecto

Cuando vuelva a ejecutar Eclipse otro día, éste por lo general recordará el proyecto en el que estuve trabajando por última vez.

Si desea trabajar en un proyecto distinto (o si desea crear un nuevo proyecto), es conveniente que primero cierre los proyectos que estén abiertos mediante la siguiente opción:

File > Close All

y después haga clic en el proyecto que desee en la parte inferior del menú **File**.

Si el nombre del proyecto no aparece en la parte inferior del menú **File**, puede localizarlo mediante el explorador de paquetes (Package Explorer), que por lo general se muestra en pantalla. Si no es así, vaya a:

Window > Show View > Package Explorer

Haga doble clic en el nivel superior del proyecto, abra la carpeta **src** y después abra **default package** para llegar a los archivos de Java. Al hacer doble clic en un archivo de java se abrirá un editor.

Como puede ver, Eclipse es muy complicado, pero es posible trabajar con un subconjunto de sus herramientas. Por desgracia, no cuenta con un modo para “principiantes”.

Cómo utilizar un editor de texto

Si desea evitar la complejidad de un IDE, puede usar un editor de texto. Tenga en cuenta que no contará con herramientas para depurar sus programas.

Hay varios editores de texto gratuitos a elegir. Por ejemplo:

Notepad++

Textpad

Crimson

Para obtenerlos, localice sus sitios Web buscando, por ejemplo, **editor crimson** en un motor de búsqueda como Google.

Estos editores de texto se pueden configurar para ejecutar el compilador e intérprete de Java con el clic de un botón, pero el método de configuración es distinto para cada editor, y puede ser complicado. Consulte sus sitios Web para obtener información detallada al respecto. En cada caso, es necesario instalar primero el sistema Java.

Otra alternativa que le ofrecemos nosotros, los autores de este libro, es un editor llamado Japa que se puede vincular con mucha facilidad al sistema Java, además de que provee un formato automático para el código. También crea plantillas que coinciden con las que utilizamos en los primeros capítulos del libro. Al igual que los otros editores, no provee herramientas de depuración. Está disponible a través del sitio Web complementario del libro.

Java en la línea de comandos

En Windows podemos abrir una ventana que nos permite escribir comandos para ejecutar Java, además de otras tareas. A esto se le conoce como símbolo del sistema, o símbolo de DOS.

En XP:

Inicio > Todos los programas > Accesorios > Símbolo del sistema

Para avanzar a una subcarpeta específica utilizamos el comando **cd** (cambiar directorio), como en el siguiente ejemplo:

```
cd misArchivos
```

Para cambiar a la carpeta que contiene la subcarpeta en la que nos encontramos, usamos:

```
cd ..
```

En Vista/Windows 7, abra el Explorador de Windows y oprima la tecla Mayús al tiempo que hace clic con el botón derecho del ratón sobre el nombre de una carpeta. Aparecerá una opción para abrir una ventana de símbolo del sistema.

Algunas veces es conveniente ejecutar programas de Java desde la línea de comandos, tal vez para pasarles parámetros de línea de comandos (argumentos) o como parte de una secuencia de comandos no interactiva.

Veamos un ejemplo. La situación es que tenemos un archivo llamado `Demo.java`. Este archivo está almacenado en la carpeta `C:\Código de Java`. El sistema de Java se instaló en:

```
C:\Archivos de programa\Java
```

Si analizamos con detalle el contenido de esta carpeta, podremos ver que el compilador (`javac.exe`) y la JVM (`java.exe`) se encuentran en una carpeta llamada `bin`. Sus nombres de archivo completos son:

```
C:\Archivos de programa\Java\jdk1.6.0_14\bin\javac.exe
C:\Archivos de programa\Java\jdk1.6.0_14\bin\java.exe
```

Localice los dos archivos en su sistema.

Para compilar un programa, abra un símbolo del sistema en la carpeta de código de Java y escriba:

```
"C:\Archivos de programa\Java\jdk1.6.0_14\bin\javac.exe" Demo.java
```

Observe que esta línea tiene dos partes:

1. La ubicación y nombre del compilador. Incluye dos espacios (entre `Archivos, de` y `programa`), por lo cual lo encerramos entre comillas.
2. El nombre del archivo que se va a compilar.

Un espacio separa los dos elementos.

Para ejecutar el archivo compilado, escribimos:

```
"C:\Archivos de programa\Java\jdk1.6.0_14\bin\java.exe" -cp . Demo
```

Aquí hay cuatro elementos:

1. La ubicación y nombre de la JVM.
2. Una bandera `-cp` (ruta de clases), la cual se utiliza para pasar datos a la JVM.
3. Un punto, que especifica el directorio actual (en este caso, `Código de Java`) como ruta de clases. Aquí es en donde la JVM buscará los archivos.
4. El nombre del archivo. No se debe utilizar la extensión `.class`.

De nuevo, hay un espacio que separa los cuatro elementos.

Argumentos de la línea de comandos

Para pasar datos al programa en ejecución mediante la línea de comandos, colocamos los elementos después del nombre del archivo, como en:

```
"C:\Archivos de programa\Java\jdk1.6.0_14\bin\java.exe" -cp . Demo "Hola a todos"
```

En este ejemplo, la cadena `Hola a todos` se pasa a `Demo`. Para ver los detalles sobre esta técnica, consulte el capítulo 17.

Cómo crear una secuencia de comandos

En definitiva, escribir comandos extensos es tedioso y propenso a errores, por lo que con frecuencia es más fácil crear una secuencia de comandos. Todos los sistemas operativos permiten esto; aquí le enseñaremos cómo se puede crear una secuencia de comandos en Microsoft Windows.

Una secuencia de comandos es un archivo que contiene comandos para ejecutar programas. Le indicamos al sistema operativo que ejecute el archivo en vez de tener que escribir cada comando. El concepto es el de un lote de comandos.

Veamos un ejemplo. Necesitamos una secuencia de comandos para compilar nuestro programa `Demo.java` anterior. Vamos a usar un editor de texto para crear un archivo que contenga las siguientes dos líneas:

```
"C:\Archivos de programa\Java\jdk1.6.0_14\bin\javac.exe" Demo.java  
pause
```

Guardamos este archivo en la misma carpeta que `Demo.java`. En Windows, el archivo debe tener una extensión `.bat`, por lo que podemos llamarle `compilar.bat`.

La instrucción `pause` se requiere debido a que podrían aparecer errores de compilación y queremos tener la oportunidad de leerlos. Esta instrucción mantiene la ventana abierta en vez de cerrarla justo después de la ejecución.

Para ejecutar el archivo podemos escribir su nombre en la ventana del símbolo del sistema, o podemos hacer doble clic en el archivo dentro del Explorador de Windows.



Índice

' (comilla sencilla) 281
- 42
-- 42
! 131, 159
!= 121
212, 214
\$ 214
% 42, 45
&& 131, 159
(double) 52
(int) 52, 215
* 42
*/ 372
.NET 451
. (punto) 16
/ 42
/* 372
/** 373
// 31, 372
: 232, 249
; 26
[] 244
() 43
\ 280
\ 323
\n 51, 233, 271, 280
\t 233, 271, 280
|| operador 131, 159
" 14, 279
+ 42, 46, 279
++ 42
< 121
<= 121
<> 229
= 41
== 117, 121
> 121
>= 121
abs 211, 473
abstract 204
accesibilidad, *vea alcance*
acceso
 aleatorio a los archivos 319
 por flujo a los archivos 319
acos 473
ActionEvent 100
ActionListener 100
actionPerformed 25, 72, 98, 100
add 18, 100, 104, 229, 235, 455, 463
addActionListener 100, 457, 461, 485
addItem 461
addItemListener 459
addListSelectionListener 471
alcance 90, 95, 111, 184, 197, 429, 508
 de clase 90
 y excepciones 307
anidamiento 163, 375, 378
 de instrucciones if 134
animación 111
aplicación xxi
 de consola 318
append 488

Apple xx
applet xxi, 2, 450, 500
APPROVE_OPTION 335
archivo 9, 318
área de texto 232, 488
args 343
argumento
 de línea de comandos 342
 vea parámetro
ArrayList 228, 229, 455
arreglos 242, 265
 bidimensionales 265
 búsqueda detallada 254
 búsqueda rápida 253
 creación de 244
 de objetos 256
 declaración de 244, 266
 initialización de 250, 270
 longitud de 247, 268
 que se pasan como parámetros 247, 269
asignación 41
asin 473
atan 473
avance paso a paso 391
AWT xxii, 15, 98, 496, 499

barra de desplazamiento 232, 490
bases de datos 448
beans 447
bibliotecas 13, 447, 454
boolean 139
booleana, variable 139
BorderLayout 456
botón 24, 457
 de opción 482
break 136
BufferedReader 320, 326, 337, 342
bug (error) 383, 397
búsqueda 238, 254
 de archivo 327, 338
 rápida 235, 253
 serial 255
ButtonGroup 482

C# 3, 452
C++ 2, 443
cadenas 278
caja
 blanca, prueba de la 388
 negra, prueba de 385
cálculo 35, 41, 210

Calendar 415, 459
campo de texto 17, 103, 492
carácter de escape 280
cargador de clases 445
carpeta 9
case 137
casilla de verificación 459
catch 304, 307, 308, 309
ChangeListener 106
char 280
charAt 287, 486
checkError 323
ciclo(s) 152
 anidados 163
 infinito 404
clases 16, 171
 abstractas 204, 423, 433
 diseño de 172
 entrada y salida de 320
 lineamientos para 371
class 175
 extensión de archivo 12
clear 234, 455
close 323, 326, 466, 482
código fuente abierto 446
Color 28, 188
comentario 31, 372
comparación de cadenas 143
compareTo 284, 486
compilación 397
compilador 10
componente 243
composición 361
concatenación 46
conurrencia 407
condiciones complejas 379
consistencia 381
constante 53, 203, 249, 269, 373
constructor 92, 96, 180, 200
 múltiple 181
 predeterminado 181
 sin parámetros 182, 200, 201
Container 463
contains 455
control deslizable 106, 485
conversión
 de números 47, 52
 de Swing a AWT 497
 de tipos 52, 215, 444
coordenadas 25
cos 211, 474
cuadro
 combinado 461

de entrada 49
 de mensaje 49
cuerpo
 de un ciclo 157
 de un método 63
currentTimeMillis 488

DataInputStream 463
Date 463
DecimalFormat 212, 463
default 138
 depuración 397, 518
 depurador 400
 desarrollo incremental 391
 desbordamiento 222
 descarga de Java 513
 destrucción 186
 diagrama
 de actividades 117, 121, 137, 157
 de clases 172, 199, 351, 355, 362, 363,
 418, 434
 directorio, *vea* carpeta
 diseño orientado a objetos 348, 349, 354,
 442
 división (operador aritmético), 42
do...while 161
 documentación 381
double 35, 36, 40
Double 47, 464
draw3DRect 468
drawArc 468
drawLine 24, 26, 27, 468
drawOval 27, 468
drawPolygon 468
drawRect 27, 468
drawRoundRect 468
drawString 468

e 37
E 37, 211, 213, 473
 E/S de consola 336
 Eclipse 9, 29, 514-519
 editor 10
 de texto 9, 519
 ejecución 11, 398
 elemento 243
 ELIZA 292
else 118, 121
 encabezado 63
 encapsulamiento 90, 177
endsWith 288, 486
 entero 35
 entorno de desarrollo integrado, *vea* IDE

 entrada
 de archivos 324
 de sitio remoto 340
equals 46, 143, 280, 283
equalsIgnoreCase 284, 486
 equivalencia 386
err 336, 338, 488
Error 310
 error de desplazamiento por uno 157
 errores comunes 401, 402, 403
 escuchar un evento 23
 especificación 384
 esquema 370
 estilo 369
 estructura de datos 228, 240, 242, 265
 estructural, prueba 388
 es-un, relación 361
 etiqueta 101, 471
 eventos 22, 72, 98, 129
 de clic del ratón 23, 479
 de desplazamiento 23
 excepción 222, 301, 303
 clases y herencia 314
 cuándo utilizar 304
 no verificada 312
 verificada 310
Exception 310
 exhaustiva, prueba 385
exists 332, 465
exit 338, 488
exp 211, 473
 expresiones 54
extends 98, 196
 extensión 9

false 139
File 331, 465
FileDialog 465
FileInputStream 466
FileNotFoundException 312
FileOutputStream 466
FileReader 326
FileWriter 323
fillArc 468
fillOval 123, 468
fillPolygon 468
fillRect 468
fillRoundRect 469
final 53, 203, 249, 373
 firma 81
FlowLayout 456, 466
 flujo 319
Flush 337, 482

for 158
 mejorado 232, 248
format 212, 464
formato
 de números 211
 de texto 51
Frame 469
Frasier 292
funcional, prueba 385

genéricos 229
get 236, 455, 459
 método 179
getAbsolutePath 332, 335, 465
getContentPane 469
getFile 465
getGraphics 477
getHeight 477
getIconHeight 469
getIconWidth 469
getName 465
getParent 332, 465
getSelectedFile 335
getSelectedIndex 461, 471
getSelectedItem 461
getSelectedValue 471
getSource 129
getText 103, 290, 488, 492
getValue 107, 485
getWidth 477
glosario 504
GNU/Linux xx
Gosling, James 2
gráficos 22, 218
Graphics 469
GUI 19

hasMoreTokens 289, 487
herencia 98, 194, 195, 199, 361
 múltiple 421
 y clases de excepciones 314
hilos 406
historia 1
HORIZONTAL 106
HTML 451, 502
IDE 9, 10, 29, 514

if 116
if..else 118
igual, operador 121, 280
IllegalArgumentException 310
ImageIcon 109, 469
implements 98, 100, 417

import 15, 94, 175, 426, 455
in 336, 488
incremento 42
independencia de la plataforma 442
indexof 287, 487
índice 231, 233, 243, 245, 265, 267
inicialización 174, 180, 250, 270, 390, 403
inicilizar 40
init 503
InputStreamReader 337, 342
insert 488
inspección 390
instalación 513
instanceof 434
instancia 17, 173, 175
instanciar 92
int 35, 36, 40
Integer 47, 103, 280, 290, 470
interface 417
interfaces
 comparación con clases abstractas 423
 e interoperabilidad 419
 múltiples 421
 y diseño 416
interfaz 100, 416
Internet xx, 2, 449
InterruptedException 408
invasores del ciberespacio 354
invocación 60, *vea también llamada a un método*
Invocación de Métodos Remotos, vea RMI
IOException 323
isAlive 495
isDirectory 332, 465
isSelected 459, 482, 483
iteración 217, *vea también repetición*

Japa 519
Java xix, 1, 2, 8, 441, 513
 beans 447
 versiones xxiii, 446
java, extensión de archivo 11
javac 520
javadoc 373
JavaScript 452
JButton 99, 457
JCheckBox 459
JComboBox 461
JDBC 448
JDK xx, 514
JFileChooser 333
JLabel 101, 471
JList 471
JMenu 474

JMenuBar 474
JMenuItem 474
join 412, 495
JOptionPane 14, 49, 338, 399, 476
JPanel 24, 104, 477
JRadioButton 482
JScrollPane 232, 491
JSlider 106, 485
JSP 450
JTextArea 488
JTextField 17, 103, 492
JVM 10, 12, 443, 445

 Kit de Herramientas de Ventanas Abstractas *vea*
 AWT

 lanzamiento de una excepción 309
lastIndexOf 288, 487
length 247, 268, 286, 333, 465, 487
 línea de comandos 342, 519
 lineamientos de diseño 365
 líneas
 en blanco 370
 extensas 43, 63
 Linux xx, 2, 8, 443
 LISP 239
list 333, 465
 lista 471
 llamada a un método 64
 llaves 26, 43, 44
log 211, 473
 lugares decimales 211

main 72, 97, 503
 Mandelbrot 226
 manejo de eventos 23, 72
 Máquina Virtual de Java, *vea* JVM
Math 53, 186, 211, 473
max 211, 473
 mayor o igual que 121
 mayor que 121
 menor o igual que 121
 menor que 121
 mensaje de error 11
 menú 474
 métodos 4, 15, 25, 60, 351
 declaración de 63
 lineamientos para 374
 nombres de 62
 Microsoft 451
min 211, 474
 modelo-vista-controlador 354
 módulo (operador aritmético), 42

mouseClicked 479
mouseDragged 481
mouseMoved 481
 movimiento del ratón 480
 multihilos 407
 multiplicación (operador aritmético) , 42

NaN 223
NegativeInfinity 223
new 17, 18, 92, 94, 97, 173, 244
nextDouble 96, 484
nextInt 92, 96, 127, 484
nextToken 289, 487
 no es igual a, 121
 no, operador 131, 159
 nombres 371
 conflictos de 71
 de clase 28
 de método 62
 reglas para 37, 175, 506
 notación de punto 16, 17
 nueva línea 51, 233, 271, 280
null 181
NullPointerException 111
NumberFormatException 307, 310, 312
 número(s)
 conversión de 52
 de punto flotante 35
 real 35

o, operador 131, 159
 objeto(s) 4, 15, 79, 88, 185
 identificación de 349, 355
 ocultamiento de información, *vea*
 encapsulamiento

openConnection 342
 operadores 41, 42
 aritméticos 42
 de comparación 154
 orientado a objetos xix, 15, 442
out 336, 488
OutOfMemoryError 310

package 427
 páginas nuevas 370
paintIcon 109, 469
 palabra clave 38, 507
 panel 24, 104, 477
 de opción 476
 paquetes 426
 paralelismo 407
 parámetro(s) 16, 26, 60, 63, 64, 66, 291
 actual 63, 66

formal 63, 66
paso de 64
paréntesis, vea llaves
parseDouble 48, 290, 464
parseInt 48, 103, 187, 280, 290, 304, 306, 469
paso a paso por instrucciones 400
pi 53
PI 53, 187, 211, 473
pixel 25
polimorfismo 432, 433
portabilidad 2, 442
PositiveInfinity 223
pow 211, 474
precedencia 42
print 323, 337, 482
println 337, 482
PrintStream 482
PrintWriter 320, 323
prioridad de hilos 413
private 64, 90, 177, 182, 184, 197, 508
 método 182
programa 3
programación a la larga 426
prompt 337
protected 195, 196, 197, 508
prueba
 de la caja blanca 388
 de la caja negra 385
 estructural 388
 exhaustiva 385
 funcional 385
public 64, 175, 177, 178, 184, 197, 508
 método 177, 197
punto de interrupción 400, 518
punto y coma 26

RAM 318
random 211, 474
Random 92, 127, 484
readLine 320, 326, 337, 463
recolección de basura 186, 445
recorrido 390
 paso a paso 400
redefinición 198, 206
refactorización 366, 371
remove 234, 455
rendimiento 443
repetición 4, 152
replace 285, 487
replaceRange 489
resta (operador aritmético) 42
resultados 73

return 73, 74
reutilizar 360
RMI 449
round 211, 215, 474
run 411
RunTimeException 310

salida de archivos 321
sangría 116, 134, 154, 164, 175, 370
ScrollingTextArea 490
secuencia 4, 29
 de comandos 520
seguridad 444
selección 4, 115
separator 465
servlets 450
set 240, 455
 método 179
setBackground 104, 477
setColor 28, 468, 469
setDelay 105, 495
setEditable 492
setFont 457, 471, 492
setLayout 463
setPreferredSize 104, 477
setSeed 484
setSize 17, 469
setTabSize 272, 489
setText 103, 281, 489, 492
ShowInputDialog 49, 477
ShowMessageDialog 14, 49, 476
ShowOpenDialog 335
ShowSaveDialog 335
sin 187, 211, 474
sitio Web xxiii
size 230, 455
sleep 408, 495
sobrecarga 80, 81, 96, 198, 206
sqrt 186, 211, 474
start 105, 411, 495
stateChanged 107
static 186, 187
 método 186
stop 105, 495
String, objetos 279, 486
 como parámetros 291
 comparación 143, 283
 conversión 289
 corrección 285
 examinar 286
StringIndexOutOfBoundsException 281
 StringTokenizer 288, 328, 487
subclase 195

subíndice 243
substring 286, 487
suma (operador aritmético), 42
Sun 2, 441
super 200
superclase 195
sustantivos 353
Swing xxii, 15, 98, 496, 499
switch 136
System 336, 488

tabulador 233, 271, 280
tan 211, 474
this 79, 184
Thread 408, 495
throw 304, 309
Throwable 310
throws 304, 309
tiene-un, relación 361
Timer 104, 495
tipo primitivo 185
toLowerCase 285, 487
toString 47, 103, 212, 290, 463, 464,
 469
toUpperCase 285, 487
trim 285, 487
true 139
try 304, 308

UML, *vea* diagrama de clases, diagrama de
actividades

Unicode 449
unión de cadenas 46
Unix 2
URL 340
URL 342
URLConnection 342
usa 172

valueOf 487
variable 35, 90
 de clase 188
 de instancia 89, 90, 177, 351
 declaración de 37
 local 70, 89, 90, 197
 private 197
 public 197
 static 187
verbos 353
verificación 384
versiones 446
VERTICAL 106
vinculación 398
visibilidad, *vea* alcance
void 64, 75
while 153

Windows 514
World Wide Web 450

y, operador 131, 159
yield 413, 494

Si es la primera vez que aborda la programación de computadoras, este libro es para usted. El texto empieza desde cero sin pedirle conocimientos previos sobre programación; además está escrito en un estilo ameno, claro y directo para una máxima comprensión.

Esta sexta edición está totalmente actualizada e incluye las nuevas características de la versión 6.0 de Java. La metodología basada en GUI de los autores facilita a los estudiantes el desarrollo de sus habilidades de programación mediante la producción de salida gráfica emocionante y dinámica.

Principales características de esta edición

- ▶ Se enfoca en las aplicaciones y el desarrollo de GUI con Swing.
- ▶ Utiliza gráficos desde los primeros capítulos, con el fin de promover el interés y el entretenimiento, además de ilustrar visualmente los principios de programación.
- ▶ Emplea UML 2.0 para el modelado y diseño.
- ▶ Incluye una gran cantidad de ejercicios y prácticas de autoevaluación con sus respectivas soluciones.

En el sitio Web de este libro (www.pearsoneducacion.net/bell) encontrará recursos adicionales para estudiantes y profesores.

ACERCA DE LOS AUTORES

Douglas Bell y **Mike Parr** tienen muchos años de experiencia enseñando programación en el Reino Unido. Han escrito varios libros sobre este tema, entre los que se encuentran *Java para estudiantes*, *Visual Basic para estudiantes* y *C# para estudiantes*. Actualmente siguen impartiendo clases y aprendiendo sobre programación, con gran entusiasmo.

Prentice Hall
es una marca de

ISBN 978-607-32-0557-3

90000



9 786073 205573

PEARSON

Visítenos en:
www.pearsoneducacion.net